PATENT
5143-01801

"EXPRESS MAIL" MAILING
LABEL NUMBER EL588940110US
DATE OF DEPOSIT JULY 14, 2000
I HEREBY CERTIFY THAT THIS PAPER OR FEE IS BEING DEPOSITED WITH THE UNITED STATES POSTAL SERVICE "EXPRESS MAIL POST OFFICE TO ADDRESSEE" SERVICE UNDER 37 C.F.R. § 1.10 ON THE DATE INDICATED ABOVE AND IS ADDRESSED TO THE COMMISSIONER OF PATENTS AND TRADEMARKS, WASHINGTON, D.C. 20231

Debra J. Tix

MEMORY MODULE INCLUDING SCALABLE EMBEDDED PARALLEL DATA COMPRESSION AND DECOMPRESSION ENGINES

By:

Thomas A. Dye
Manuel J. Alvarez II
Peter Geiger

Attorney Docket No.: 5143-01801

Jeffrey C. Hood
Conley, Rose & Tayon, P.C.
P.O. Box 398
Austin, Texas 78767-0398
Ph: (512) 476-1400

Title: Memory Module Including Scalable Embedded Parallel Data Compression and Decompression Engines

Inventors: Thomas A. Dye, Manuel J. Alvarez II, and Peter Geiger

Assignee: Interactive Silicon, Incorporated, Austin, Texas

Priority Claim

This application claims benefit of priority of U.S. Provisional Application Serial No. 60/144,125 titled "Memory Module Including Scalable Embedded Parallel Data Compression and Decompression Engines", filed July 16, 1999, whose inventors are Thomas A. Dye, Manuel J. Alvarez II, and Peter Geiger

Continuation Data

This is a continuation-in-part (CIP) of U.S. patent application Serial No. 09/239,659 titled "Bandwidth Reducing Memory Controller Including Scalable Embedded Parallel Data Compression and Decompression Engines" and filed January 29, 1999, whose inventors are Thomas A. Dye, Manuel J. Alvarez II, and Peter Geiger.

Field of the Invention

The present invention relates to computer system and memory module device architectures, and more particularly to a memory module which includes an embedded data compression and decompression engine for the reduction of data bandwidth and improved efficiency.

Description of the Related Art

Since their introduction in 1981, the architecture of personal computer systems has remained substantially unchanged. The current state of the art in computer system architectures includes a central processing unit (CPU) that couples to a memory controller interface that in turn couples to system memory. The computer system also includes a separate graphical interface for coupling to the video display. In addition, the computer

system includes input/output (I/O) control logic for various I/O devices, including a keyboard, mouse, floppy drive, hard drive, etc.

In general, the operation of modern computer architecture is as follows. Programs and data are read from a respective I/O device such as a floppy disk or hard drive by the operating system, and the programs and data are temporarily stored in system memory. Once a user program has been transferred into the system memory, the CPU begins execution of the program by reading code and data from the system memory through the memory controller. The application code and data are presumed to produce a specified result when manipulated by the system CPU. The CPU processes the code and data, and data is provided to one or more of the various output devices. The computer system may include several output devices, including a video display, audio (speakers), printer, etc. In most systems, the video display is the primary output device.

Graphical output data generated by the CPU is written to a graphical interface device for presentation on the display monitor. The graphical interface device may simply be a video graphics array (VGA) card, or the system may include a dedicated video processor or video acceleration card including separate video RAM (VRAM). In a computer system including a separate, dedicated video processor, the video processor includes graphics capabilities to reduce the workload of the main CPU. Modern prior art personal computer systems typically include a local bus video system based on the Peripheral Component Interconnect (PCI) bus, the Advanced Graphics Port (AGP), or perhaps another local bus standard. The video subsystem is generally positioned on the local bus near the CPU to provide increased performance.

Therefore, in summary, program code and data are first read from the hard disk to the system memory. The program code and data are then read by the CPU from system memory, the data is processed by the CPU, and graphical data is written to the video RAM in the graphical interface device for presentation on the display monitor.

The system memory interface to the memory controller requires data bandwidth proportional to the application and system requirements. Thus, to achieve increased system performance, either wider data buses or higher speed specialty memory devices are

required. These solutions force additional side effects such as increased system cost, power and noise.

The CPU typically reads data from system memory across the local bus in a normal or non-compressed format, and then writes the processed data or graphical data back to the I/O bus or local bus where the graphical interface device is situated. The graphical interface device in turn generates the appropriate video signals to drive the display monitor. It is noted that prior art computer architectures and operation typically do not perform data compression and/or decompression during the transfer between system memory and the CPU or between the system memory and the local I/O bus. Prior art computer architecture also does nothing to reduce the size of system memory required to run the required user applications or software operating system. In addition, software controlled compression and decompression algorithms typically controlled by the CPU for non-volatile memory reduction techniques cannot be applied to real time applications that require high data rates such as audio, video, and graphics applications. Further, CPU software controlled compression and decompression algorithms put additional loads on the CPU and CPU cache subsystems.

Certain prior art systems utilize multiple DRAM devices to gain improved memory bandwidth. These additional DRAM devices may cost the manufacturer more due to the abundance of memory that is not fully utilized or required. The multiple DRAM devices are in many instances included primarily for added bandwidth, and when only the added bandwidth is needed, additional cost is incurred due to the multiple DRAM packages. For example, if a specific computer system or consumer computing appliance such as a Digital TV set-top box uses DRDRAM memory and requires more than 1.6Gbytes/sec of bandwidth, then the minimum amount of memory for this bandwidth requirement will be 16Mbytes. In such a case, the manufacture pays for 16 Mbytes even if the set-top box only requires 8 Mbytes.

Computer systems are being called upon to perform larger and more complex tasks that require increased computing power. In addition, modern software applications require computer systems with increased graphics capabilities. Modern software applications include graphical user interfaces (GUIs) that place increased burdens on the graphics

capabilities of the computer system. Further, the increased prevalence of multimedia applications also demands computer systems with more powerful graphics capabilities. Therefore, a new system and method is desired to reduce the bandwidth requirements required by the computer system application and operating software. A new system and method is desired which provides increased system performance without specialty high-speed memory devices or wider data I/O buses required in prior art computer system architectures.

Database Systems Background

It has been estimated that 95% of all commercial client/server environments are built around database systems. These environments are usually constructed to accommodate a large number of users performing a large number of sophisticated database queries and operations to a large distributed database. These compute, memory and I/O intensive environments put great demands on database servers. If a database client or server is not properly balanced, then the number of database transactions per second that it can process can drop dramatically. A system is considered balanced for a particular application when the PROCESSOR tends to saturate about the same time as the I/O subsystem.

Ideally, a client should not notice any substantial degradation in response time for a given transaction even as the number of transactions requested per second by other clients to the database server increases. The size of main memory plays a critical role in a database servers ability to scale for this application. In general, a database server will continue to scale up until the point that the application data no longer fits in main memory. Beyond this point, the buffer manager resorts to swapping pages between main memory and disks. The amount of disk paging increases exponentially as a function of the fraction of main memory available causing application performance and response time to degrade exponentially as well. At this point, the application is said to be I/O bound.

When a user performs a sophisticated data query, thousands of pages may be needed from the database, which is typically distributed across many disks, and possibly distributed across many systems. To minimize the overall response time of the query, access times must be minimized to any database pages that are referenced more than once, as well as, the

enormous amount of temporary data this is generated by the server. If the buffer cache is not large enough, then many of those pages will have to be repeatedly fetched and swapped to disk.

Independent studies have shown that when 70 to 90% of the working data fits in main memory, most applications will run several times slower. When only 50% of the working data fits in main memory, most applications run 5 to 20 times slower. Typical relational database operations run 4 to 8 times slower when only 66% of the working data fits in main memory. The need to reduce or eliminate disk paging is compelling. Unfortunately, for system designers, the demand for more main memory by database applications will continue to far exceed the rate of advances in memory density. Cost effective methods are needed to increase the effective size of system memory.

It is difficult for I/O bound applications to take advantage of recent advances in processor, cache and system memory performance improvements since they are constrained by the high latency and low bandwidth of disk subsystems. The most common way to reduce disk paging is to add memory. Adding memory to database servers can get expensive since these applications demand a lot of memory. Alternatively, disk request and data bandwidth can be increased by adding more disks and disk caches. It may be even necessary to move to a larger server with multiple, higher performance I/O buses. Memory and disks are added until the database server becomes balanced. Main memory compression is a smart alternative to adding more memory and I/O for many reasons.

First, main memory compression increases the effective size of main memory by compressing and storing a large block of data into a smaller space. The effective size of main memory is typically doubled. This allows client/server applications, which typically work on data much larger than main memory, to execute more efficiently due to the decreased number of disk requests. The number of disk requests is reduced because pages that have been accessed before are more likely to still be in main memory when accessed again due to the increased capacity of main memory.

Today, large numbers of disks are added to the system to satisfy the high disk request rates generated by client/server applications. As a result, it is common that only a fraction of the disk space on each disk is utilized. By effectively reducing the disk request

rate, fewer disk caches are needed to queue the requests, and fewer disk drives are needed to serve these requests. In addition, due to the reduced number of disks needed, the disk space associated with each disk can be more fully utilized.

In addition, by reducing the size of data to be transferred between local and remote disks and main memory, the I/O buses are utilized less. This reduced I/O bandwidth can be used to scale system performance beyond what it was originally capable of, or allows the I/O subsystem to be cost reduced based on the reduced amount of bandwidth needed.

Disks usually become fragmented over time. When it does, it is common for a page, or group of pages, to be written back to disk using two or more non-contiguous sectors. By reducing the size of data to be written to disk, many pages that would normally occupy two or more sectors on disk may only need to occupy one sector. As a result, the number of disk seeks should be reduced.

Since disk accesses are dominated by the latency associated with the seek time, it is not obvious that smaller blocks provide meaningful overall latency reducing benefits. However, smaller blocks should improve disk cache hit rates.

Today, data compression is used for many purposes. Application-specific software and hardware data compression is being used on audio, video, images, and many other objects. Even with today's media-rich content, the majority of the data consumed is still, and will continue to be, text and binary. By applying our compression methods in a more general manner, we will be transparent and complement other existing software and hardware compression methods.

There are major performance and cost differences between data compression of main memory and data compression of disk. Disk compression has been around a long time and has primarily been used to reduce disk storage requirements. However, it does not apply to commercial client/server applications since only a small fraction of each disk is used. For commercial applications, disks are added to the system to increase the number of disk requests that can be performed simultaneously.

Data Compression Background

Software-based data compression has been successfully used to compress and decompress very specific data types in main memory for I/O bound applications where the processor is only minimally utilized. It has been known to improve application performance by a factor of two. However, its use is limited to lightweight applications due to the processor overhead required to compress and decompress each data type. Also, software and hardware-based data compression has been used in disk subsystems. However, as previously described, its benefits are very limited.

In general, as data compression bandwidth and latency approaches the performance of main memory, and the methods used to manage compressed data becomes easier and more transparent to operating system and application software, the closer it can be to the processor, or rather, the higher it can be located in the memory hierarchy. The closer data compression can be to the processor, the more impact it can have on system performance. Starting at the top, the typical memory hierarchy comprises the L1 cache, L2 cache, application space located in main memory, buffer cache located in main memory, disk cache, and disk. For each level in the memory hierarchy, the access times and throughputs are dramatically different. The L1 and L2 caches are managed on a cache line basis, and their access times are around 1 and 6 processor clocks, respectively. Unfortunately, it is unlikely that a single cache line of data can be compressed enough to merit adding hardware compression this high up in the memory hierarchy.

Application space and buffer cache are managed on a 4KB page basis. It is well known that an LZ-based data compression algorithm is a good general-purpose algorithm that typically compresses main memory pages to about half their original size or better. The processor performs random accesses to application space. Unfortunately, when 4KB compressed pages are accessed randomly, latency increases substantially since, statistically, half the block has to be decompressed before obtaining the requested data. This probably precludes using hardware compression for application space for now. Fortunately, the buffer cache is accessed on a page basis, and the processor usually accesses buffer cache pages starting at the beginning of the 4KB page. For now, the buffer cache is probably the highest point in the memory hierarchy that hardware compression can be applied.

Independent studies have shown that hardware-based main memory compression can theoretically improve system performance by an order of magnitude or more depending on where it is located in the system. There are several possible places in a client/server system where compression hardware could be added such as the PCI bus, the AGP port, a memory DIMM, inside the memory controller, on the host bus, or inside the processor.

Today, pages are transferred between buffer cache and application space by the processor in about 10us or up to 100K pages per second. At the other extreme, pages are typically read from disk to buffer cache in about 10ms or up to 100 pages per second, three orders of magnitude slower. If a disk cache is added, then page hits can be read in about 500us or up to 2K pages per second. Using the processor, software compression can be used to transfer pages in 250us or up to 4K pages per second. Compression hardware available today compresses up to 100MB/s that may make them good candidates for the PCI bus. Page transfers based on compression hardware located on the PCI bus have about the same throughput as software compression. However, there is very little processor overhead. If hardware compression were located on the AGP port, transfer rates would improve to about 40us or up to 25K pages per second. However, AGP slots are not generally available in client/server systems

Summary of the Invention

The present invention includes parallel data compression and decompression technology, referred to as "MemoryF/X", designed for the reduction of data bandwidth and storage requirements and for compressing / decompressing data at a high rate. The MemoryF/X technology may be included in any of various devices, including a memory controller; memory modules; a processor or CPU; peripheral devices, such as a network interface card, modem, IDSN terminal adapter, ATM adapter, etc.; and network devices, such as routers, hubs, switches, bridges, etc., among others.

In one embodiment, the present invention comprises a memory module which includes the MemoryF/X technology to provide improved data efficiency and bandwidth and reduced storage requirements. The memory module includes a compression/decompression engine, preferably parallel data compression and decompression slices, that are embedded into the memory module. Further, the memory module may not require specialty memory components or system software changes for operation.

The MemoryF/X Technology reduces the bandwidth requirements while increasing the memory efficiency for almost all data types within the computer system or network. Thus, conventional standard memory components can achieve higher bandwidth with less system power and noise than when used in conventional systems without the MemoryF/X Technology.

The MemoryF/X Technology is designed to embed into a memory module or memory control circuits and has a novel architecture to compress and decompress parallel data streams within the computing system. In addition, the MemoryF/X Technology has a "scalable" architecture designed to function in a plurality of memory configurations or compression modes with a plurality of performance requirements.

The MemoryF/X Technology's system level architecture reduces data bandwidth requirements and thus improves memory efficiency. Compared to conventional systems, the MemoryF/X Technology obtains equivalent bandwidth to conventional architectures that use wider buses, specialty memory devices, and/or more attached memory devices. Both power and noise are reduced, improving system efficiency. Thus, systems that are

sensitive to the cost of multiple memory devices, size, power and noise can reduce costs and improve system efficiency.

Systems that require a minimum of DRAM memory but also require high bandwidth do not need to use multiple memory devices or specialty DRAM devices in a wider configuration to achieve the required bandwidth when the MemoryF/X technology is utilized. Thus, minimum memory configurations can be purchased that will still achieve the bandwidth required by high-end applications such as video and graphics.

As mentioned above, according to the present invention the MemoryF/X Technology includes one or more compression and decompression engines for compressing and decompressing data within the system. In the preferred embodiment, the MemoryF/X Technology comprises separate compression and decompression engines. In an alternate embodiment, a single combined compression/decompression engine can be implemented. The MemoryF/X Technology primarily uses a lossless data compression and decompression scheme.

Where the MemoryF/X Technology is included in a memory module, data transfers to and from the memory module can thus be in either two formats, these being compressed or normal (non-compressed). The memory module may also include one or more lossy compression schemes for audio/video/graphics data. Thus, compressed data from system I/O peripherals such as the non-volatile memory, floppy drive, or local area network (LAN) are decompressed in the memory module and stored into the memory module or saved in the memory module in compressed format. Thus, data can be saved in either a normal or a compressed format, retrieved from the memory for CPU usage in a normal or compressed format, or transmitted and stored on a medium in a normal or compressed format.

To improve latency and reduce performance degradations normally associated with compression and decompression techniques, the MemoryF/X Technology may encompass multiple novel techniques such as: 1) parallel lossless compression/decompression; 2) selectable compression modes such as lossless, lossy or no compression; 3) priority compression mode; 4) data cache techniques; 5) variable compression block sizes; 6) compression reordering; and 7) unique address translation, attribute, and address caches.

Where the MemoryF/X Technology is included in a memory module, one or more of these modes may be controlled by a memory controller coupled to the memory module(s).

The MemoryF/X Technology preferably includes novel parallel compression and decompression engines designed to process stream data at more than a single byte or symbol (character) at one time. These parallel compression and decompression engines modify the single stream dictionary based (or history table based) data compression method described by Lempel and Ziv to provide a scalable, high bandwidth compression and decompression operation. The parallel compression method examines a plurality of symbols in parallel, thus providing greatly increased compression performance.

The MemoryF/X Technology can selectively use different compression modes, such as lossless, lossy or no compression. Thus, in addition to lossless compression/decompression, the memory module also can include one or more specific lossy compression and decompression modes for particular data formats such as image data, texture maps, digital video and digital audio. The MemoryF/X technology may selectively apply different compression/decompression algorithms depending on one or more of the type of the data, the requesting agent, or a memory address range. In one embodiment, internal memory controller mapping allows for format definition spaces (compression mode attributes) that define the compression mode or format of the data to be read or written.

The MemoryF/X Technology may use a priority compression and decompression mode that is designed for low latency operation. In the priority compression format, memory address blocks assigned by the operating system for uncompressed data are used to store the compressed data. Hence, data-path address translation is not necessary, which optimizes bandwidth during data transfers. This also allows use of the MemoryF/X Technology with minimal or no changes to the computer operating system. Thus, for priority memory transfers, memory size is equivalent to that of data storage for non-compressed formats. The excess memory space resulting from the compression is preferably allocated as overflow storage or otherwise is not used. Thus the priority mode optimizes data transfer bandwidth, and may not attempt to reduce utilized memory.

The compression / decompression engine in the MemoryF/X Technology may use multiple data and address caching techniques to optimize data throughput and reduce

latency. The MemoryF/X Technology includes a data cache, referred to as the L3 data cache, which preferably stores most recently used data in an uncompressed format. Thus, cache hits result in lower latency than accesses of data compressed in the system memory. The L3 data cache can also be configured to store real time data, regardless of most recently used status, for reduced latency of this data.

The MemoryF/X Technology may dynamically (or statically) allocate variable block sizes based on one or more of data type, address range and/or requesting agent for reduced latency. In general, a smaller block size results in less latency than a larger block size, at the possible expense of lower compression ratios and/or reduced bandwidth. Smaller block sizes may be allocated to data with faster access requirements, such as real time or time sensitive data. Certain data may also be designated with a "no compression" mode for optimum speed and minimal latency.

The MemoryF/X Technology also includes a compression reordering algorithm to optimally reorder compressed data based on predicted future accesses. This allows for faster access of compressed data blocks. During decompression, the longest latency to recover a compressed portion of data in a compressed block will be the last symbol in the portion of the data being accessed from the compressed block. As mentioned above, larger compression block sizes will increase latency time when the symbol to be accessed is towards the end of the compressed data stream. This method of latency reduction separates a compression block at intermediate values and reorders these intermediate values so that the portions most likely to be accessed in the future are located at the front of the compressed block. Thus the block is reordered so that the segment(s) most likely to be accessed in the future, e.g. most recently used, are placed in the front of the block. Thus, these segments can be decompressed more quickly. This method of latency reduction is especially effective for program code loops and branch entry points and the restore of context between application subroutines. This out of order compression is used to reduce read latency on subsequent reads from the same compressed block address.

The MemoryF/X Technology in an alternate embodiment reduces latency further by use of multiple history windows to context switch between decompression operations of different requesting agents or address ranges. A priority can be applied such that

compression and decompression operations are suspended in one window while higher priority data is transferred into one of a number of compression / decompression stages in an alternate window. Thus, reduction of latency and improved efficiency can be achieved at the cost of additional parallel history window buffers and comparison logic for a plurality of compression / decompression stages.

The MemoryF/X Technology includes an address translation mode for reduction of memory size. This reduction of memory size is accomplished at the cost of higher latency transfers than the priority compression mode, due to the address translation required. An address translation cache may be utilized for the address translation for reduced latency. An internal switch allows for selection of priority mode compression, normal mode compression, or no compression transfers. An attribute or tag field, which in-turn may be controlled by address ranges on a memory page boundary, preferably controls the switch.

In one embodiment, the operating system, memory controller driver or BIOS boot software allocates memory blocks using a selected compression ratio. Thus, the allocated memory block size is based on a compression ratio, such as 2:1 or 4:1. Hence, the allocated block size assumes the data will always compress to at least the smaller block size.

The MemoryF/X Technology also accounts for overflow conditions during compression. Overflow occurs when the data being compressed actually compresses to a larger size than the original data size, or when the data compresses to a smaller size than the original data, but to a larger size than the allocated block size. The MemoryF/X Technology handles the overflow case by first determining whether a block will overflow, and second storing an overflow indicator and overflow information with the data. The memory controller preferably generates a header stored with the data that includes the overflow indicator and overflow information. Thus the directory information is stored with the data, rather than in separate tables. Compression mode information may also be stored in the header with the data. The MemoryF/X Technology thus operates to embed directory structures directly within the compressed data stream.

The MemoryF/X Technology also includes a combined compression technique for lossy compression. The combined compression technique performs lossless and lossy

compression on data in parallel, and selects either the lossless or the lossy compressed result depending on the degree of error in the lossy compressed result.

The integrated data compression and decompression capabilities of the MemoryF/X Technology remove system bottlenecks and increase performance, allowing lower cost systems due to smaller data storage requirements and reduced bandwidth requirements. This also increases system bandwidth and hence increases system performance. Thus, the present invention provides a significant advance over the operation of current devices, such as memory controllers, memory modules, processors, and network devices, among others.

Brief Description of the Drawings

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

Figure 1 illustrates a prior art computer system architecture;

Figure 2a illustrates a computer system including an integrated memory controller (IMC) according to the present invention;

Figure 2b illustrates a computer system having at least one memory module including the MemoryF/X Technology according to one embodiment of the present invention;

Figure 3 illustrates the internal architecture of the IMC according to the preferred embodiment;

Figure 4 is a block diagram illustrating the internal architecture of the Memory Controller unit of the IMC;

Figure 5 is a more detailed block diagram illustrating the compression / decompression logic comprised in the IMC 140;

Figure 6A illustrates the sequential compression technique of the prior art dictionary-based LZ serial compression algorithm;

Figure 6B illustrates the parallel compression algorithm according to the present invention;

Figure 7 is a high level flowchart diagram illustrating operation of the parallel compression;

Figure 8 is a more detailed flowchart diagram illustrating operation of the parallel compression;

Figure 9 illustrates the entry data history and input data compare and results calculation for the parallel compression and decompression unit;

Figure 10 shows the parallel selection and output generation block diagram;

Figure 11a shows the operation of the counter values, output counter and output mask used for output selection during the parallel compression operation of the present invention;

Figures 11b and 11c are tables which show the operation of the counter values, output counter and output mask used for output selection during the parallel compression operation of the present invention;

Figure 11d is a table that illustrates the generation of the combined mask from the collection of output masks;

Figure 12 illustrates the Output Generator Flow diagram;

Figure 13 illustrates an example of the parallel compression operation indicating the data flow through multiple cycles;

Figure 14 illustrates a high-speed parallel comparison circuit used to find the largest count of matching entries to the history table;

Figure 15 further illustrates the select generation logic and entry compare logic designed for high data clocking rates;

Figure 16 illustrates the logic table for the high-speed parallel comparison;

Figure 17 illustrates the lossy compression and decompression engines;

Figure 18 is a table that shows the lossy compression output format for image data that does not include alpha values;

Figure 19 is a table that shows the lossy compression output format for image data that includes alpha values;

Figure 20 is a block diagram of the combination lossy and lossless compression and decompression operation;

Figure 21 illustrates a plurality of compression formats for source and destination data as used by the IMC for compression and decompression memory efficiency;

Figures 22 and 23 are flowchart diagrams illustrating operation of memory accesses using the compression mode features of the present invention;

Figure 24 illustrates the flow for compression address translation, dictionary and overflow block address translation;

Figure 25 is a table illustrating the memory allocation fields for the compression allocation table and the Overflow table, compression memory area and the overflow memory area;

Figure 26 illustrates the initialization process flow for the compression address translation table;

Figure 27 illustrates the store transaction process flow for the compression and decompression unit;

Figure 28 illustrates the memory fetch process flow;

Figure 29 illustrates the next address generation process flow;

Figure 30 is a table illustrating the memory allocation space and compression ratios according to one implementation of the present invention;

Figure 31 illustrates the compression re-ordering algorithm use to reduce read data latency of subsequent memory read cycles by requesting system agents;

Figure 32 is a table illustrating the header information presented to the lossless decompression engine;

Figure 33 illustrates the four stages used for the parallel lossless decompression algorithm;

Figure 34 illustrates the eight decoder stages required to generate the start counts used for the parallel decompression process;

Figure 35 illustrates a single decoder block used by the stage 1 input selector and byte counter of Figure 33;

Figure 36a is a table indicating the check valid results table of the decode block;

Figure 36b is a table describing the Data Generate outputs based on the Data Input and the Byte Check Select logic;

Figure 37 illustrates a portion of the second of the four stages illustrated in Figure 33 for calculating selects and overflows according to one embodiment of the invention;

Figure 38 illustrates a portion of the third of the four stages illustrated in Figure 33 for converting preliminary selects generated in stage two into final selects according to one embodiment of the invention;

Figure 39 illustrates a portion of the fourth of the four stages illustrated in Figure 33 for generating uncompressed output bytes from selects generated in the first three stages according to one embodiment of the invention;

Figure 40 illustrates the data flow through the parallel lossless decompression engine according to one embodiment of the invention;

Figure 41 illustrates an embodiment with three decoder stages to accept 32 bits of input data and generate the information used for the parallel decompression process;

Figure 42a illustrates a decompression engine with four input bytes, three decoders, and four output bytes according to one embodiment of the invention;

Figure 42b illustrates an example decompression of an input to the decompression engine illustrated in Figure 42b according to one embodiment of the invention;

Figure 43a is a high-level flowchart of the operation of a parallel decompression engine;

Figure 43b is a flowchart illustrating a parallel decompression method according to one embodiment of the invention;

Figure 43c is a flowchart illustrating a process for examining a plurality of tokens from the compressed data in parallel according to one embodiment of the invention;

Figure 43d is a flowchart illustrating a process for extracting one or more tokens to be decompressed in parallel according to one embodiment of the invention;

Figure 43e is a flowchart illustrating a process for generating count and index or data byte information in parallel according to one embodiment of the invention;

Figure 43f is a flowchart illustrating a process for generating a plurality of selects to symbols in a combined history window according to one embodiment of the invention;

Figure 43g is a flowchart illustrating a process for generating preliminary selects according to one embodiment of the invention;

Figure 43h is a flowchart illustrating a process for generating final selects according to one embodiment of the invention;

Figure 43i is a flowchart illustrating a process for writing uncompressed symbols from the combined history window to the output data according to one embodiment of the invention;

Figure 43j is a flowchart illustrating a process for writing symbols uncompressed by the current decompression cycle to the history window according to one embodiment of the invention; and

Figure 43k is a flowchart illustrating a decompression process combining Figures 43b, 43c and 43d according to one embodiment of the invention;

Figures 44a and 44b illustrate a memory module including memory components, e.g., SDRAM components, and the MemoryF/X Technology according to one embodiment of the present invention; and

Figures 45-48 illustrate various aspects of memory module compression / decompression.

Detailed Description of the Preferred Embodiment

Incorporation by Reference

U.S. patent application Serial No. 09/239,659 titled "Bandwidth Reducing Memory Controller Including Scalable Embedded Parallel Data Compression and Decompression Engines" and filed January 29, 1999, whose inventors are Thomas A. Dye, , Manuel J. Alvarez II, and Peter Geiger.

U.S. Patent No. 5,838,334 titled "Memory and Graphics Controller which Performs Pointer-Based Display List Video Refresh Operations", whose inventor is Thomas A. Dye, and which issued on November 17, 1998, is hereby incorporated by reference in its entirety as though fully and completely set forth herein.

U.S. Patent Application Serial No. 08/340,667 titled "Integrated Video and Memory Controller with Data Processing and Graphical Processing Capabilities" and filed November 16, 1994, whose inventor is Thomas A. Dye, is hereby incorporated by reference in its entirety as though fully and completely set forth herein.

U.S. Patent Application Serial No. 08/463,106 titled "Memory Controller Including Embedded Data Compression and Decompression Engines" is hereby incorporated by reference in its entirety as though fully and completely set forth herein.

U.S. Patent Application Serial No. 09/056,021 titled "Video / Graphics Controller Which Performs Pointer-Based Display List Video Refresh Operations" and filed June 17, 1998, whose inventor is Thomas A. Dye, is hereby incorporated by reference in its entirety as though fully and completely set forth herein.

The present invention includes parallel data compression and decompression technology, referred to as "MemoryF/X", designed for the reduction of data bandwidth and storage requirements and for compressing / decompressing data at a high rate. The MemoryF/X technology may be included in any of various devices, including a memory controller; memory modules; a processor or CPU; peripheral devices, such as a network

interface card, modem, IDSN terminal adapter, ATM adapter, etc.; and network devices, such as routers, hubs, switches, bridges, etc., among others.

In a first embodiment, the present invention comprises a system memory controller, referred to as the Integrated Memory Controller (IMC), which includes the MemoryF/X technology. The IMC is discussed in U.S. patent application Serial No. 09/239,659 titled "Bandwidth Reducing Memory Controller Including Scalable Embedded Parallel Data Compression and Decompression Engines" and filed January 29, 1999, referenced above.

In a second embodiment, the present invention comprises a memory module that includes the MemoryF/X technology to provide improved data efficiency and bandwidth and reduced storage requirements. The memory module includes a compression/decompression engine, preferably parallel data compression and decompression slices, that are embedded into the memory module. Further, the memory module may not require specialty memory components or system software changes for operation.

In a third embodiment, the present invention comprises a network device, such as a router, switch, bridge, or hub, which includes the MemoryF/X technology of the present invention. The network device can thus transfer data in the network at increased speeds and/or with reduced bandwidth requirements.

The MemoryF/X Technology reduces the bandwidth requirements while increasing the memory efficiency for almost all data types within the computer system or network. Thus, conventional standard memory components can achieve higher bandwidth with less system power and noise than when used in conventional systems without the MemoryF/X Technology.

The technology of the present invention is described below with reference to a computer system architecture, which is one example of the use of the present invention. Thus, Figures 1 and 2 illustrate example embodiments, and it is noted that the technology described herein may be included in any of various systems or architectures. The following describes the operation of the MemoryF/X Technology being incorporated into either a memory controller or a memory module. However, the MemoryF/X Technology may be incorporated into any of various devices.

Example Computer System Architecture

Figure 1 illustrates a block diagram of an example computer system architecture. As shown, computer architectures typically include a CPU 102 coupled to a cache system 104. The CPU 102 couples to the cache system 104 and couples to a local bus 106. A memory controller 108, referred to as North Bridge 108, is coupled to the local bus 106, and the memory controller 108 in turn couples to system memory 110. The graphics adapter 112 is typically coupled to a separate local expansion bus such as the peripheral component interface (PCI) bus or the Accelerated Graphics Port (AGP) bus. Thus, the north-bridge memory controller 108 is coupled between the CPU 102 and the main system memory 110 wherein the north-bridge logic also couples to the local expansion bus where the graphics adapter 112 is situated. The graphics adapter 112 may couple to frame buffer memory 114 that stores the video data, also referred to as pixel data, that is actually displayed on the display monitor. Modern computer systems typically include between 1 to 8 Megabytes of video memory. Alternatively, the video or pixel data may reside in the system memory, and frame buffer memory may not be included, as shown in Figure 2a. An I/O subsystem controller 116 is shown coupled to the local bus 106. In computer systems that include a PCI bus, the I/O subsystem controller 116 typically is coupled to the PCI bus. The I/O subsystem controller 116 couples to a secondary input/output (I/O) bus 118. Various peripheral I/O devices are generally coupled to the I/O bus 18, including a non-volatile memory, e.g., hard disk 120, keyboard 122, mouse 124, and audio digital-to-analog converter (DAC) 238.

According to one embodiment of the present invention, the system memory modules 110 include the MemoryF/X Technology of the present invention. The frame buffer memory modules 114 may also include the MemoryF/X Technology of the present invention. The system memory modules 110 thus comprise memory components or devices as well as the MemoryF/X Technology, which includes parallel compression / decompression engines. The MemoryF/X Technology is operable to compress /

decompress data as it is transferred to / from the memory components or devices comprised on the module. Similarly, the frame buffer memory modules 114 may comprise memory components or devices as well as the MemoryF/X Technology.

The memory components or devices comprised on the memory modules 110 and/or 114 may be any of various types, such as an SDRAM (static dynamic random access memory) DIMM (dual in-line memory module) or other types of memory components. In addition, specialty technology such as RAMBUS can be used both in the memory device and memory control unit to supply high bandwidth at low pin count. For more information on the RAMBUS memory architecture, please see "RAMBUS Architectural Overview," version 2.0, published July 1993 by RAMBUS, Inc., and "Applying RAMBUS Technology to Desktop Computer Main Memory Subsystems," version 1.0, published March 1992 by RAMBUS, Inc., which are both hereby incorporated by reference.

In another embodiment of the present invention, the North Bridge 108 includes the MemoryF/X Technology of the present invention. In another embodiment of the present invention, the MemoryF/X Technology is distributed between the North Bridge 108 and the memory modules 110.

Integrated Memory Computer Architecture

Figure 2a is a block diagram illustrating another embodiment of a system architecture incorporating the present invention. Elements in Figure 2a that are similar or identical to those in Figure 1 include the same reference numerals for convenience.

As shown, the computer system of the present invention includes a CPU 102 preferably coupled to a cache system 104. The CPU 102 may include an internal first level cache system and the cache 104 may comprise a second level cache. Alternatively, the cache system 104 may be a first level cache system or may be omitted as desired. The CPU 102 and cache system 104 are coupled to a Local bus 106. The CPU 102 and cache system 104 are directly coupled through the Local bus 106 to an integrated memory controller (IMC) 140 according to the present invention.

The integrated memory controller (IMC) 140 performs memory control functions. It is noted that the IMC 140 can be used as the controller for main system memory 110 or can

be used to control other memory subsystems as desired. The IMC 140 couples to system memory 110, wherein the system memory 110 comprises one or more banks of DRAM memory and may comprise a plurality of different type memory devices. The IMC 140 includes a memory controller core.

The memory modules 110 of the present invention may use of various types of memory components, as desired. In the preferred embodiment, the memory modules comprise SDRAM DIMMs. In another embodiment, the memory modules are based on a RAMBUS implementation. For more information on the RAMBUS memory architecture, please see the RAMBUS references mentioned above, which were incorporated by reference. In an alternate embodiment, the system memory 110 comprises SGRAM or single in-line memory modules (SIMMs). As noted above, the memory modules 110 may use any of various types of memory components, as desired.

The IMC 140 may also generate appropriate video signals for driving video display device 142. The IMC 140 preferably generates red, green, blue (RGB) signals as well as vertical and horizontal synchronization signals for generating images on the video display 142. Therefore, the integrated memory controller 140 may integrate memory controller and video and graphics controller capabilities into a single logical unit. This greatly reduces bus traffic and increases system performance. The IMC 140 may also generate appropriate data signals that are provided to Audio DAC 238 for audio presentation. Alternatively, the IMC 140 integrates audio processing and audio DAC capabilities and provides audio signal outputs that are provided directly to speakers.

The IMC 140 is situated on either the main CPU bus or a high-speed system peripheral bus. The IMC 140 may also be closely or directly integrated with the CPU 102, e.g., comprised on the same chip as the CPU 102. As shown in Figure 2a, the IMC 140 may be coupled directly to the Local bus 106 or CPU bus, wherein the IMC 140 interfaces through a L2 cache system 104 to the CPU 102. In an alternate embodiment, the L2 cache and controller 104 may be integrated into the CPU 102 or into the IMC 140, or not used.

An I/O subsystem controller 116 is coupled to the Local bus 106. The I/O subsystem controller 116 in turn is coupled to an optional I/O bus 118. Various I/O devices are coupled to the I/O bus including a non-volatile memory, e.g., hard disk 120, keyboard

122, and mouse 124, as shown. In one embodiment, the I/O bus is the PCI bus, and the I/O subsystem Controller 116 is coupled to the PCI bus.

The IMC 140 may include a high level protocol for the graphical manipulation of graphical data or video data, which greatly reduces the amount of bus traffic, required for video operations and thus greatly increases system performance. This high level protocol includes a display list based video refresh system and method whereby the movement of objects displayed on the video display device 142 does not necessarily require movement of pixel data in the system memory 110, but rather only requires the manipulation of display address pointers in a Display Refresh List, thus greatly increasing the performance of pixel bit block transfers, animation, and manipulation of 2D and 3D objects. For more information on the video/graphics operation of the IMC 140, please see U.S. Patent No. 5,838,334. The IMC 140 also includes an improved system and method for rendering and displaying 3D objects.

According to one embodiment, as described above, the system memory modules 110 include the MemoryF/X Technology of the present invention. The system memory modules 110 thus comprise memory components or devices as well as the MemoryF/X Technology, which includes parallel compression / decompression engines. The operation of the compression/decompression logic is discussed in greater detail below. In another embodiment of the present invention, the IMC 140 includes the MemoryF/X Technology of the present invention. In another embodiment of the present invention, the MemoryF/X Technology is distributed between the IMC 140 and the memory modules 110.

Main memory DRAM devices at the 64-Mbit levels and higher continue to increase the package sizes and number of address and data pins. The increased pin count due to this trend eliminates the ability to "bank" DRAMS for higher effective bandwidth as in smaller DRAM architectures of the past. In addition, to lower effective bandwidth the "wide" DRAM devices cost more to manufacture due to increased package cost, test equipment, and testing time. In order to increase bandwidth, the system memory controller and/or the memory modules must be designed with additional I/O data pins to compensate for wider DRAM devices, resulting in higher power usage and noise.

For computer appliances that require minimum main memory configuration and also require high bandwidth, the current choices are currently limited to specialty high speed memory devices such as RAMBUS or DDRDRAM which cost more, consume more power and generate more noise, or multiple smaller DRAM packages that typically require more PC board real-estate. The MemoryF/X Technology of the present invention is operable to provide high bandwidth with simplified and inexpensive memory components.

Figure 2b is a block diagram illustrating one embodiment of a system, wherein the MemoryF/X Technology 200 is comprised on at least one memory module 110. One or more of the system memory modules 110 thus may comprise memory components or devices as well as the MemoryF/X Technology, which includes one or more parallel compression / decompression engines. The MemoryF/X Technology is operable to compress / decompress data as it is transferred to / from the memory components or devices comprised on the module.

One or more of the frame buffer memory modules 114 in Figure 2B may also include the MemoryF/X Technology of the present invention. In a similar manner the one or more frame buffer memory modules 114 may comprise memory components or devices as well as the MemoryF/X Technology.

The memory components or devices comprised on the memory modules 110 and/or 114 may be any of various types, such as an SDRAM (static dynamic random access memory) DIMM (dual in-line memory module) or other types of memory components. In addition, specialty technology such as RAMBUS can be used both in the memory device and memory control unit to supply high bandwidth at low pin count. For more information on the RAMBUS memory architecture, please see "RAMBUS Architectural Overview," version 2.0, published July 1993 by RAMBUS, Inc., and "Applying RAMBUS Technology to Desktop Computer Main Memory Subsystems," version 1.0, published March 1992 by RAMBUS, Inc., which are both hereby incorporated by reference.

In another embodiment of the present invention, the MemoryF/X Technology may be distributed between the memory controller, e.g., the North Bridge 108 or the IMC 140, and one or more of the memory modules 110.

The following describes embodiments of the present invention, wherein the MemoryF/X Technology is incorporated either into a memory controller, e.g., the IMC 140, or in memory modules, e.g., memory modules 110. Figures 3 - 5 illustrate an embodiment wherein the MemoryF/X Technology is incorporated into the IMC 140. Figures 6 onward generally describe the operation of the MemoryF/X Technology.

Figure 3 – IMC Block Diagram

Figure 3 is a block diagram illustrating the internal components comprising the IMC 140 in the preferred embodiment. In this embodiment, the IMC 140 may incorporates the MemoryF/X Technology according to the present invention. As shown, the IMC 140 integrates a data compression/decompression engine and control functions into the memory controller unit 220. This reduces the amount of non-volatile (disk) storage or archive storage requirements and reduces the amount of bandwidth required to move data in the system, and thus reduces overall system costs. This also reduces the required amount of system memory because, when data is compressed for storage, more non-recently-used or off-screen data can be stored in system memory 110.

It is noted that the present invention may be incorporated into any of various types of computer systems or devices having various system architectures. In alternate embodiments of the present invention, the data compression/decompression engine can be integrated into any device that connects to memory. In some embodiments, the present invention improves bandwidth and efficiency without increase in cost to the system or increased I/O bus requirements.

The memory controller may operate in different compression modes. One mode, referred to as normal compression mode, reduces the amount of memory used by translating addresses allocated by the operating system into new addresses which minimize the memory usage according to the compression that is performed. While this embodiment may reduce the amount of memory used, an alternate mode, referred to as priority compression mode, does not make use of memory size savings and instead trades off the

additional saved memory for higher bandwidth and lower overall latency. In the priority compression mode, no changes to the software or operating system software are necessary (other than initialization code) to implement the compression / decompression improvements. The normal and priority compression modes are discussed below.

It is noted that various of the elements in Figure 3 are interconnected with each other, wherein many of the various interconnections are not illustrated in Figure 3 for simplicity.

As shown, the IMC 140 includes bus interface logic 202 for coupling to the host computer system, for coupling to the Local bus 106. In the preferred embodiment, the Local bus 106 is the CPU bus or host bus. Alternatively, the Local bus 106 is the PCI bus, and the bus interface logic 202 couples to the PCI bus. Instruction storage/decode logic (not shown) may be coupled to the bus interface logic 202.

The bus interface logic 202 couples to the memory control unit 220. The MemoryF/X technology preferably resides internal to the memory controller block 220. A control bus 201 connects all units to the local CPU interface 202. An execution engine 210 is coupled through the control bus 201 to the local CPU interface 202 and the memory interface 221 and the execution engine 210 also couples to the memory controller. Local bus 106 data and commands are routed through the local CPU interface to the control bus 201 which in turn is coupled to the execution engine 210, the memory interface 221, the graphics engine 212, the Peripheral I/O bus interface 234, the VDRL engine 240, a video input and format conversion unit 235 and finally the audio & modem subsystem 236. In addition the execution engine 210 is coupled to the main system memory 110 through the memory controller 220 and the memory interface 221.

The graphics engine 212 is also coupled to the main system memory 110 through the memory controller 220 and the memory interface 221. Thus, data is read and written for rasterization and pixel draw output by the graphics engine 212 with assistance for data transfer and efficiency by the memory controller 220. In addition, the other blocks are coupled under similar circumstances through the memory controller 220 and memory interface 221 to the system memory 110.

As shown in Figure 3 the memory controller 220 transfers data between the system memory 110 and the requesting units. The requesting units include the execution engine 210, local CPU or RISC interface 202, audio and modem subsystem 236, Video I/O interface 235, VDRL engine 240, peripheral bus interface 234 and graphics engine 212. The requesting units will request the memory controller 220 for data transfer operations to the system memory 110 through the system memory interface 221. Each requesting unit may represent or utilize a different compression format, allowing higher memory efficiency. Thus, there are pluralities of data compression formats under control of the requesting units and supported by the memory controller block 220.

Figure 4 - Memory Controller Unit

Figure 4 illustrates the memory controller block 220. In the preferred embodiment the memory controller 220 includes a parallel compression and decompression engine 251. In an alternate embodiment, the memory controller 220 includes a single or serial compression engine and a single or serial decompression engine. In addition, in the preferred embodiment, the parallel compression and decompression unit 251 includes a separate lossy compression and decompression engine (discussed later in this disclosure) which also may be designed as separate or unified units. Additional alternate embodiments may apply individual compression and/or decompression units located in multiple areas of the IMC 140 for optimal efficiency of compression or decompression.

The memory controller block 220 may include one or more parallel or serial compression/decompression engines, including one or more parallel and/or serial lossless compression/decompression engines and/or one or more parallel and/or serial lossy compression/decompression engines. The term "compression/decompression engine" as used herein is intended to include all such combinations of one or more parallel, serial, lossless and/or lossy compression/decompression engines, whether they be integrated or separate blocks, and whether they be comprised in or external to the memory controller, or comprised in another unit, such as the CPU 102.

Support blocks for the preferred embodiment of the memory controller 220 preferably include the switch logic 261, compression control unit 281, compressed data

directory 271, L3 data cache memory 291, and the memory interface logic 221. Main system memory 110 in Figure 4 is preferably external to the memory controller block 220 and is shown only for reference. In addition, the L3 data cache 291 may also be standard memory (SRAM or Embedded DRAM) in absence of external memory and may be configured other than as cache type memory. Input signals to the memory controller 220 preferably comprise a request bus and control bus 211, and a plurality of address buses 215 and data buses 216 from each requesting unit in the IMC 140 as indicated in Figure 4. Alternatively, each of the requesting agents may share common address/data buses. The memory controller 220 generates output signals that interface to the main system memory 110. These output signals comprise a plurality control signals required to drive multiple DRAM memory devices as previously indicated.

Again referring to Figure 4, the switch logic 261 preferably interfaces to all the requesting unit's address and data buses, including control buses and strobes necessary to indicate valid data and address cycles presented to the memory controller 220. The switch logic 261 also includes the necessary ports to drive address and data to the other units within the memory controller 220. The switch logic 261 controls read and write data to and from the parallel compression and decompression unit 251 and the compression control unit 281. In addition, for data that is not to be compressed or decompressed (normal or bypass data), the switch logic 261 controls an interface directly to the memory interface logic 221. In order to properly control the switching direction of the address and data for different data compression formats, the switch logic 261 receives control inputs from the compression control unit 281 and the Request bus 211. The switch logic 261 also interacts with the parallel compression and decompression unit 251 as described in detail later. Thus, the switch logic 261 arbitrates the incoming requests for memory control and data transfer operations, ranking requests in a priority scheme and filtering the requests for normal or compressed memory transactions.

Again referring to Figure 4, the compression control unit 281 receives memory transaction requests from the request and control bus 211 and receives addresses from the switch unit 261 for control of each memory transaction. The compression control unit 281 directs the switch logic 261, the compression data directory 271, the local data cache

memory (L3 data cache) 291, the memory interface logic 221, and the parallel compression and decompression unit 251 for proper operation and set-up for each memory transaction request. The compression control unit 281 interfaces to the compressed data directory 271. The compressed data directory 271 is used for look up of the address block start location for one of the L3 data cache 291, the SRAM buffers (located in the Parallel Compression and Decompression unit 251) or the system memory 110. Thus, the compression control unit 281 receives requests from other units in the IMC 140, translates the location by address, determines the compression block size, and controls the sub-units of the memory controller 220 for the proper address and data transactions as required to read or write data to and from the main system memory 110.

The data cache 291 shown in Figure 4 is used to minimize the latency of operation by returning requested data that has been recently used. The data cache 291 is an L3 data cache where the CPU 102 or system includes L1 and L2 caches. The cache 291 may also operate as an L2 or L1 cache for the CPU 102, as desired. The cache 291 is referred to as an L3 cache in this description.

The L3 data cache size will determine the average number of clocks required to return data to the requesting units of the IMC 140. In the present embodiment, most recently used data is stored in a non-compressed format in the L3 data cache 291. For data that resides in the L3 data cache 291, no compression or decompression action is required by the parallel compression and decompression unit 251. Thus, a transaction request with an L3 data cache hit can return data with less latency than a transaction request that requires a main memory 110 transaction. The L3 data cache 291 typically contains only uncompressed data, although in alternate embodiments the L3 cache 291 may store most recently used data in a compressed format, or in a combination of compressed and non-compressed formats. Thus, the L3 data cache 291 located in the memory controller 210 can return most recently used data without the normal latency delay associated with conventional memory controllers.

In one embodiment where the parallel compression and decompression engine 251 does not contain SRAM buffer storage, the L3 data cache 291 can double for such SRAM buffers used to store write blocks for future compression and read blocks for future

decompression. Thus, the L3 data cache 290 may be used to store compressed blocks that await future decompression for either read or write operations. For example, the L3 data cache 291 may be used to store LRU pages that are waiting to be compressed and transferred to the non-volatile memory. Thus the L3 data cache 291 and associated cache control logic 281 buffer the transactions to improve memory access latency for both read and write operations of both compressed/decompressed transactions or transactions that require uncompressed operation (no compression or decompression).

Again referring to Figure 4, the memory interface logic 221 receives control signals form the compression control unit, receives address and data from either the switch logic 261 (non-compressed transactions), or the compression data directory 271 and controls the timing and delivery voltage levels to the main memory 110 depending on the DRAM device type. Thus, the memory interface logic 221 is used to interface to the main system memory 110 matching the memory configuration and device type.

The Parallel compression and decompression unit 251 is described in detail in the following sections.

Figure 5 - Compression/Decompression Engine

As shown in Figure 5, the parallel compression and decompression 251 block preferably includes compression engines 570/575 and decompression engines 550/555. As noted above, the parallel compression and decompression unit 251 may contain a single lossless parallel compression and decompression engine and/or a single lossy compression and decompression engine, or a combination of lossless and/or lossy engines.

The parallel compression and decompression unit 251 performs high-speed parallel compression and decompression using a parallel symbol data stream, instead of a serial symbol data stream as in conventional implementations. The parallel operation of the compression and decompression unit 251 is optimized for bandwidth reduction and reduced latency. Thus the parallel compression and decompression engines allows a higher speed decompression and compression rate, which substantially increases bandwidth and reduces latency of that over prior art compression and decompression engines. The algorithm for the parallel compression invention is further described in detail below.

Figure 5 also illustrates the internal diagram of the switch logic 261. The switch 261 performs data format and address conversion as well as the arbitration of multiple requests from a plurality of other units in the IMC 140. The switch logic 261 includes a crossbar switch 502 that performs the selection of the current memory transaction request. This selection is performed by one of a plurality of arbitration methods with the intention to deliver data first to units that must operate real time memory transactions. In the preferred embodiment, the order of priority for such requesting units is first the display refresh requests from the VDRL engine 240, followed by the Video I/O unit 235, the Audio and Modem 236, the Local CPU/RISC interface 202, the Graphics engine 212 and execution engine 210, followed by the Peripheral I/O bus interface 234. The priority order, block size, and request latency is software programmable by the interface driver software for the IMC 140. Thus, the system performance and memory transaction efficiency and/or response can be adjusted dynamically by software control executed by the interface drivers. Such interface software is preferably executed on the CPU 102 but alternatively can be executed by the execution engine 210.

The switch logic 261 preferably contains specific data selection units separating normal uncompressed reads and writes from compressed reads and writes. Decompression switch 512 determines a block read operation by sending command, address, block tags, data type and length information to the decompression engine 550 and 555. In addition, the decompression switch 512 receives decompressed data and transaction tag information from the decompression engine 550 and/or 555. The decompression switch 512 is preferably pipelined for a plurality of system memory read requests at the same time. The tag field allows multiple outstanding requests to be issued to the decompression engines 550 and/or 555 in parallel.

Similarly, the switch logic 261 contains a normal memory switch 514 for read and write transactions that require no compression or decompression operation. In the preferred embodiment, some data address ranges or requests from specific request units may not need or want to have compression operations. Thus, the memory switch 514 generates block transfer, address generation, data tags, length, and command information for interface to the memory interface unit 560.

The switch logic 261 includes compress switch 516 that performs command, address, tag, length and data type preparation for the compression engine 570 and/or 575. Data written to the memory controller 220 by a plurality of requesting units 211 are received by the compress switch 516 and will be either compressed and written to main memory 110 or, if in the valid address range of the L3 data cache 291, will be written to the L3 data cache 291 under control of the memory switch 514.

Thus, the compression cache control unit 281 along with the switch unit 261 determine the transaction type, priority and control required to complete the transaction by either the L3 data cache 291, the parallel compression and decompression unit 251 or the main memory interface 560. As indicated in Figure 5, the preferred embodiment shows transaction sizes of 16 data bytes. In alternate embodiments, the transaction sizes can be any number of data bytes.

As discussed above in Figure 4, the L3 data cache 291 interacts with the cache control unit 281. For transactions that have address ranges with associated data located within the L3 data cache 291, the decompression engine 550, memory interface 560, and compression engine 570, are not used, and data is read or written directly into the L3 data cache 291. Thus, for L3 data cache 291 hits, data bypasses the parallel compression and decompression unit 251 and is read or written directly to/from the L3 data cache 291 in a non-compressed format.

In addition, again referring to Figure 5, the parallel compression and decompression unit 251 includes data and command transfer multiplexers 522 and write data multiplexers 590. The command transfer multiplexers 522 perform data, command address, tag, length switching, and interfacing to the decompression engine 550/555, memory interface 560, and compression engines 570/575. Alternate embodiments may include the transfer multiplexers 522 in the switch logic 261 in a single rather than multiple bus design. The write data multiplexers 590 perform the selection between normal (uncompressed) data writes and compressed data writes to the main memory 110.

The memory interface unit 221 interfaces to the decompression engines 550 and/or 555 for status, tags and read data, interfaces to the memory interface 560 for both read, write control, address and tags, and interfaces to the compression engines 570 and/or 575 for

write data. The memory interface unit 221 includes a DRAM controller 592 and a DRAM I/O interface 594. The DRAM controller 592 performs the timing of the control signals and address to the DRAM I/O interface 594 to control the main memory bank 110. In the preferred embodiment the control of RDRAM memory is controlled by the high speed analog RAC located within the DRAM I/O interface 594. In alternate embodiments, other memory types such as SDRAM, DRDRAM, SLDRAM, or VMC require additional logic in the DRAM I/O interface 594. Thus, the memory interface logic 221 is internal to the memory controller 220 and interfaces to the compression control unit 281 for control signals, the switch logic 261 for address, tags, control and data signals, the parallel compression and decompression unit 251 for address, control and data transactions. In addition, the memory interface logic 221 performs the memory interface and signal conditioning for interfacing to the main system memory 110.

Parallel Lossless Compression and Decompression

The parallel compression/decompression unit or engine 251, which performs parallel compression and decompression functions, is now discussed. The engine 251 is preferably a dedicated codec hardware engine, e.g., the engine is comprised of logic circuitry. In one embodiment, the codec engine 251 comprises a programmable DSP or CPU core, or programmable compression/decompression processor, with one or more ROMs or RAMs which store different sets of microcode for certain functions, such as compression, decompression, special types of graphical compression and decompression, and bit blit operations, as desired. In this embodiment, the codec engine 251 dynamically shifts between the different sets of microcode in the one or more memories, depending on the function being performed. The compression/decompression engine may also be implemented using reconfigurable or programmable logic, e.g., one or more FPGAs.

As shown in Figure 5, in one embodiment, the engine 251 preferably includes an embedded lossless parallel data compression engine 570 and parallel decompression engine 550 designed to compress and decompress data as data is transferred to/from system memory 110. The compression engine 570 and decompression engine 550 may be constructed using any of the techniques described with reference to the engine 251,

including hardware engines comprised of logic circuitry, programmable CPUs, DSPs, a dedicated compression/decompression processor, or reconfigurable or programmable logic, to perform the parallel compression and decompression method of the present invention. Various other implementations may be used to embed a compression/decompression within the memory controller according to the present invention. In the preferred embodiment, the compression engine 570 and decompression engine 550 comprise hardware engines in the IMC 140, or alternatively use pieces of the same engine for compression and decompression. In the following description, the parallel compression and decompression unit is described as having separate compression and decompression engines 570 and 550.

For a general overview of the benefits and methods for using compression and decompression engines in the main system memory controller, refer to US patent disclosure titled "Memory Controller Including Embedded Data Compression and Decompression Engines", filed June 5, 1995, serial number 08/463,106, whose inventor is Thomas A. Dye.

Thus, the IMC 140 includes two data formats referred to as "compressed" data and "non-compressed" data. The compressed data format requires less storage and thus is less expensive. The compressed format also requires less system bandwidth to transfer data between system memory 110 and I/O subsystems. The decompression from compressed data format to normal data format results in a small performance penalty. However, the compression of non-compressed data format to compressed data format does not have an associated penalty, although there may be an added latency that would normally be hidden. However, if the data doesn't compress well, and there is a long series of stores that need compressed, the bus could be backed up causing read and snoop delays to the processor. In one embodiment, the compression engine 570 is implemented in software by the CPU 102.

In the preferred embodiment, the compression engine 570 and decompression engine 550 in the IMC 140 comprise one or more hardware engines that perform a novel parallel lossless compression method, preferably a "parallel" dictionary based compression and decompression algorithm. The parallel algorithm may be based on a serial dictionary based algorithm, such as the LZ77 (preferably LZSS) dictionary based compression and decompression algorithm. The parallel algorithm may be based on any variation of

conventional serial LZ compression, including LZ77, LZ78, LZW and/or LZRW1, among others.

The parallel algorithm could also be based on Run Length Encoding, Predictive Encoding, Huffman, Arithmetic, or any other lossless compression algorithm. However, the parallelizing of these is less preferred due to their lower compression capabilities and/or higher hardware costs.

As a base technology, any of various lossless compression methods may be used as desired. As noted above, a parallel implementation of LZSS compression is preferably used, although other lossless compression methods may allow for fast parallel compression and decompression specifically designed for the purpose of improved memory bandwidth and efficiency.

For more information on a data compression and decompression system using serial LZ compression, please see U.S. Patent No. 4,464,650 which is hereby incorporated by reference. The above patent presents implementations of the LZ77 data compression method described by Lempel and Ziv in "Compression of Individual Sequences Via Variable-Rate Coding," IEEE Transactions on Information Theory, IT-5, September 1977, pages 530-537, and "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, Volume 23, No. 3 (IT-23-3), May 1977, pages 337-343, wherein the above two articles are both hereby incorporated by reference. U.S. Patent No. 4,701,745, titled "Data Compression System," which issued October 20, 1987, describes a variant of LZ77 called LZRW1, and this patent is hereby incorporated by reference in its entirety. A modified version of the LZ78 algorithm is referred to as LZW and is described in U.S. Patent No. 4,558,302. Another variant of LZW compression is described in U.S. Patent No. 4,814,746.

In an alternate embodiment, the data compression and decompression engines 570 and 550 utilize parallel data compression/decompression processor hardware based on the technology disclosed in U.S. Patent No. 5,410,671, titled "Data Compression / Decompression Processor," which issued April 25, 1995 and which is hereby incorporated by reference in its entirety.

The IMC 140 may also utilize parallel data compression/decompression techniques of the present invention based on the serial techniques described in U.S. Patent No. 5,406,279 titled "General Purpose, Hash-Based Technique for Single Pass Lossless Data Compression,"; U.S. Patent No. 5,406,278 titled "Method and Apparatus for Data Compression Having an Improved Matching Algorithm which Utilizes a Parallel Hashing Technique,"; and U.S. Patent No. 5,396,595 titled "Method and System for Compression and Decompression of Data." In alternate embodiments, other types of parallel or serial data compression/decompression methods may be used.

The compression/decompression engine 251 of the present invention may include specialized compression/decompression engines 575/555 for image data. The preferred embodiment of the lossy compression/decompression engine is described with reference to Figures 17 - 20, and a parallel version is described with reference to Figures 32 - 36.

Other embodiment may utilize image compression and decompression techniques shown and described in U.S. Patent No. 5,046,119 titled "Method and Apparatus for Compressing and Decompressing Color Video Data with an Anti-Aliasing Mode," this patent being hereby incorporated by reference in its entirety. For related information on compression and decompression engines for video applications, please see U.S. Patent No. 5,379,356 titled "Decompression Processor for Video Applications," U.S. Patent No. 5,398,066 titled "Method and Apparatus for Compression and Decompression of Digital Color Images," U.S. Patent No. 5,402,146 titled "System and Method for Video Compression with Artifact Disbursement Control," and U.S. Patent No. 5,379,351 titled "Video Compression/Decompression Processing and Processors," all of which are hereby incorporated by reference in their entirety.

Figure 6A - Prior Art

Prior art has made use of the LZ compression algorithm for design of computer hardware, but the bandwidth of the data stream has been limited due to the need to serially review the incoming data to properly generate the compressed output stream. Figure 6A depicts the prior art normal history table implementation.

The LZ compression algorithm attempts to reduce the number of bits required to store data by searching that data for repeated symbols or groups of symbols. A hardware implementation of an LZ77 algorithm would make use of a history table to remember the last n symbols of a data stream so that they could be compared with the incoming data. When a match is found between the incoming stream and the history table, the matching symbols from the stream are replaced by a compressed symbol, which describes how to recover the symbols from the history table.

Figure 6B - Parallel Algorithm

The preferred embodiment of the present invention provides a parallel implementation of dictionary based (or history table based) compression / decompression. By designing a parallel history table, and the associated compare logic, the bandwidth of the compression algorithm can be increased many times. This specification describes the implementation of a 4 symbol parallel algorithm which results in a 4 times improvement in the bandwidth of the implementation with no reduction in the compression ratio of the data. In alternate embodiments, the number of symbols and parallel history table can be increased and scaled beyond four for improved parallel operation and bandwidth, or reduced to ease the hardware circuit requirements. In general, the parallel compression algorithm can be a 2 symbol parallel algorithm or greater, and is preferably a multiple of 2, e.g., 2, 4, 8, 16, 32, etc. The parallel algorithm is described below with reference to a 4 symbol parallel algorithm for illustrative purposes.

The parallel algorithm comprises paralleling three parts of the serial algorithm: the history table (or history window), analysis of symbols and compressed stream selection, and the output generation. In the preferred embodiment the data-flow through the history table becomes a 4 symbol parallel flow instead of a single symbol history table. Also, 4 symbols are analyzed in parallel, and multiple compressed outputs may also be provided in parallel. Other alternate embodiments may contain a plurality of compression windows for decompression of multiple streams, allowing a context switch between decompression of individual data blocks. Such alternate embodiments may increase the cost and gate counts with the advantage of suspending current block decompression in favor of other block

decompression to reduce latency during fetch operations. For ease of discussion, this disclosure will assume a symbol to be a byte of data. Symbols can be any reasonable size as required by the implementation. Figure 6B shows the data-flow for the parallel history table.

Figure 7 - High Level Flowchart of the Parallel Compression Algorithm

Figure 7 is a high-level flowchart diagram illustrating operation of the parallel compression algorithm in the preferred embodiment. Steps in the flowchart may occur concurrently or in different orders.

In step 402, the method maintains a history table (also called a history window) comprising entries, wherein each entry may comprise one symbol. The history table is preferably a sliding window that stores the last n symbols of the data stream.

In step 404 the method maintains a current count of prior matches which occurred when previous symbols were compared with entries in the history table. A count is maintained for each entry in the history table.

It is noted that maintenance of the history table and the current counts are performed throughout the algorithm based on previously received symbols, preferably starting when the first plurality of symbols are received for compression.

In step 406, the method receives uncompressed data, wherein the uncompressed data comprises a plurality of symbols. Thus, the parallel compression algorithm operates on a plurality of symbols at a time. This is different from conventional prior art serial algorithms, which operate in a serial manner on only one symbol at a time. The plurality of symbols comprises two or more symbols, preferably a power of two. In the preferred embodiment, the parallel compression algorithm operates on four symbols at a time. However, implementations using 8, 16, 32 or more symbols, as well as other non-power of 2 numbers, may be readily accomplished using the algorithm described herein.

In step 408, the method compares the plurality of symbols with each entry in the history table in a parallel fashion. This comparison produces compare results. Each entry in the history table preferably compares with each of the plurality of symbols concurrently, i.e., in a parallel fashion, for improved speed.

In step 410, the method determines match information for each of the plurality of symbols based on the current count and the compare results. Step 410 of determining match information includes determining zero or more matches of the plurality of symbols with each entry in the history table. More specifically, step 410 may include determining a longest contiguous match based on the current count and the compare results, and then determining if the longest contiguous match has stopped matching. If the longest contiguous match has stopped matching, then the method resets or updates the current counts.

As noted above, step 410 also includes resetting the counts for all entries if the compare results indicate a contiguous match did not match one of the plurality of symbols. The counts for all entries are preferably reset based on the number of the plurality of symbols that did not match in the contiguous match. In the preferred embodiment, the method generates a reset value for all entries based on the compare results for a contiguous match. The reset value indicates a number of the plurality of symbols that did not match in the contiguous match as indicated in the compare results. The method then updates the current counts according to the compare results and the reset value.

In step 412 the method outputs compressed data information in response to the match information. Step 412 may involve outputting a plurality of sets of compressed data information in parallel, e.g., for different matches and/or for non-matching symbols. Step 412 includes outputting compressed data information corresponding to the longest contiguous match that stopped matching, if any. The contiguous match may involve a match from a prior plurality of symbols. Step 412 may also include outputting compressed data information solely from a prior match. Step 412 also includes, for non-matching symbols that do not match any entry in the history table, outputting the non-matching symbols in an uncompressed format.

For a contiguous match, the compressed data information includes a count value and an entry pointer. The entry pointer points to the entry in the history table that produced the contiguous match, and the count value indicates a number of matching symbols in the contiguous match. In one embodiment, an encoded value is output as the count value,

wherein more often occurring counts are encoded with fewer bits than less often occurring counts.

Steps 402 - 412 are repeated one or more times until no more data is available. When no more data is available, then, if any current counts are non-zero, the method outputs compressed data for the longest remaining match in the history table.

Since the method performs parallel compression, operating on a plurality of symbols at a time, the method preferably accounts for symbol matches comprised entirely within a given plurality of symbols, referred to as the "special case". Here presume that the plurality of symbols includes a first symbol, a last symbol, and one or more middle symbols. Step 410 of determining match information includes detecting if at least one contiguous match occurs with one or more respective contiguous middle symbols, and the one or more respective contiguous middle symbols are not involved in a match with either the symbol before or after the respective contiguous middle symbols. If this condition is detected, then the method selects the one or more largest non-overlapping contiguous matches involving the middle symbols. In this instance, step 412 includes outputting compressed data for each of the selected matches involving the middle symbols.

Figure 8 - Detailed Flowchart of the Parallel Compression Algorithm

Figure 8 is a more detailed flowchart diagram illustrating operation of the parallel compression algorithm in the preferred embodiment. Steps that are similar or identical to steps in Figure 7 have the same reference numerals for convenience.

In the flowchart of Figure 8, it is presumed that the method maintains a history table comprising entries, wherein each entry comprises one symbol. The history table is preferably a sliding window that stores the last n symbols of the data stream. It is also presumed that the method maintains a current count of prior matches that occurred when previous symbols were compared with entries in the history table. A count is maintained for each entry in the history table. As noted above, the maintenance of the history table and the current counts are performed throughout the algorithm, preferably starting when the first plurality of symbols are received for compression.

In step 406, the method receives uncompressed input data, wherein the uncompressed data comprises a plurality (or group) of symbols. Thus, the parallel compression algorithm operates on a plurality of symbols at a time. This is different from conventional prior art algorithms, which operate in a serial manner on only one symbol at a time. The plurality of symbols comprises two or more symbols, preferably four symbols. As noted above, the parallel compression algorithm can operate on any number of symbols at a time. The input data may be the first group of symbols from a data stream or a group of symbols from the middle or end of the data stream.

In step 408, the method compares the plurality of symbols with each entry in the history table in a parallel fashion. This comparison produces compare results. Each entry in the history table preferably compares with each of the plurality of symbols concurrently, i.e., in a parallel fashion, for improved speed.

In step 422, the method determines zero or more matches of the plurality of symbols with each entry in the history table. In other words, in step 422 the method determines, for each entry, whether the entry matched any of the plurality of symbols. This determination is based on the compare results.

If no matches are detected for the plurality of symbols in step 422, then in step 432 the method determines if any previous matches existed. In other words, step 432 determines if one or more ending symbols from the prior group of symbols matched entries in the history table, and compressed information was not yet output for these symbols since the method was waiting for the new plurality of symbols to possibly determine a longer contiguous match. If one or more previous matches existed as determined in step 432, then in step 434 the method outputs the previous compressed data information. In this case, since the prior matches from the prior group of symbols are not contiguous with any symbols in the current group, the previous compressed data information is output. After step 434, operation proceeds to step 436.

If no previous matches existed as determined in step 432, or after step 434, then in step 436 the method outputs each symbol of the plurality of symbols as uncompressed symbols. Since each of the plurality of symbols does not match any entry in the history table, then each of the plurality of symbols are output in an uncompressed format. After

step 436, in step 438 all counters are reset to 0. In step 472 the uncompressed symbols are added to the history window, and operation returns to step 406 to receive more input data, i.e., more input symbols.

If one or more matches are detected for the plurality of symbols in step 422, then in step 442 the method determines if all of the plurality of symbols is comprised in one match. If so, then in step 444 the method increases the count for the respective entry by the number of matching symbols, e.g., four symbols. In step 474, the uncompressed symbols are added to the history window, and operation returns to step 406 to receive more input data, i.e., more input symbols. In this case, the method defers providing any output information in order to wait and determine if any symbols in the next group contiguously match with the current matching symbols.

If not all of the plurality of symbols is comprised in one match as determined in step 442, then in step 452 the method determines if any previous matches existed. The determination in step 452 is similar to the determination in step 432, and involves determining if one or more ending symbols from the prior group of symbols matched entries in the history table, and compressed information was not yet output for these symbols since the method was waiting for the new plurality of symbols to possibly determine a longer contiguous match.

If one or more previous matches existed as determined in step 452, then in step 454 the method selects the largest contiguous match including the previous match. In step 456, the method outputs compressed data information regarding the largest contiguous match. This compressed data information will include previous compressed data information, since it at least partly involves a previous match from the previous group of symbols. If the first symbol in the current plurality of symbols is not a contiguous match with the previous match, then the compressed data information will comprise only the previous compressed data information. After step 456, operation proceeds to step 462.

Steps 462 - 470 are performed for each input symbol in a parallel fashion. In other words, steps 462 - 470 are performed concurrently for each input symbol. Steps 462-470 are shown in a serial format for ease of illustration.

In step 462, the method determines if the respective symbol is included in any match. If not, then in step 464 the method outputs the uncompressed symbol. In this case, the respective symbol does not match any entry in the history table, and thus the symbol is output uncompressed.

If the respective symbol is included in a match as determined in step 462, then in step 466 the method determines if the match includes the last symbol. If not, then in step 468 the method outputs compressed data information for the match. It is noted that this may involve a "special case" involving a match comprising only one or more middle symbols.

If the match does include the last symbol as determined in step 466, then in step 470 the method resets counters to the maximum of the symbol count in the match. In this case, compressed information is not output for these symbols since the method waits for the new plurality of symbols to possibly determine a longer contiguous match.

Once steps 462 - 470 are performed for each input symbol in parallel, then in step 472 the uncompressed symbols are added to the history window. Operation then returns to step 406 to receive more input data, i.e., a new plurality or group of input symbols. If no more input data is available or is received, then in step 480 the method flushes the remaining previous matches, i.e., provides compressed information for any remaining previous matches.

The method of Figure 8 also accounts for matches within the middle symbols as described above.

Figures 9 and 10 - Operation of the Parallel Compression Algorithm

Figures 9 and 10 are hardware diagrams illustrating operation of the parallel compression algorithm. As with the prior art LZ serial algorithm, each entry of the history table contains a symbol (byte) of data, which is compared with the input stream of data 610. The input stream 610 comprises Data0, Data1, Data2 and Data3. Figure 9 illustrates an entry of the history table, referred to as entry D 602. As shown, entry D 602 is compared with each symbol of the input stream 610. Figure 9 illustrates Entry D 602 of the parallel implementation, and its inputs and outputs. Comparators 608 compare each data byte entry

with the four bytes from the input stream 610, and generate four compare signals (labeled D0 through D3 for entry D). Compare signal D0 is used in entry D. The compare signal D1 will be used by the next entry E in the history table, compare signal D2 will be used by entry F, and compare signal D3 will be used by entry G. Accordingly, entry D uses compare signal 3 from entry A, 2 from compare signal entry B and code 1 from entry C. These can be seen as inputs to the results calculation block 606 in Figure 9. The results of this compare are held in a counter 604 that is part of the entry logic. The counter values are sent to the compressed stream selection logic 612/614/616 (Figure 10) to determine if the input data is being compressed or not. This information is forwarded to the output generation logic 618 that sends either the uncompressed data to the output, or the compressed stream data.

An embodiment of the generation of the Output Mask and Output count from the results calculation block 606, along with the Entry Counter update value, is described in the table of Figure 11a. The New Counter Value is calculated by counting the number of matches that occur beginning with A3 and continuing to D0. For example, an A3 and B2 match without a C1 match sets the counter to two. The special case of all four compares matching adds 4 to the present counter value.

Generation of the counter output is similar, comprising the Saved counter (counter value prior to the setting of the new counter value) plus the count of matches starting with D0 and continuing to A3. The output mask is generated by inverting the 4 match signals and adding a 5^{th} signal which is 1 for all cases except for a special case of a C1 and B2 match without a D0 or an A3 match. This special case allows the compression of the two bytes centered in the input word. The Reset Value will be generated by the selection logic 612 from the mask value. The reset value is included in this disclosure as indicated in the table of Figure 11a for ease of description only.

An embodiment of the generation of the Output Mask from the results calculation block 606, along with the Counter update value and the Entry Maximum Count Flag, is described in the table of Figure 11b. The New Counter Value is calculated by counting the number of matches that occur beginning with A3 and continuing to D0. For example, an A3 and B2 match without a C1 match sets the counter to 2. The special case of all four compares matching adds 4 to the present counter value.

The output mask is an encoded value based on the matches that have occurred in this entry, and the maximum count flag for this entry. The tables of Figures 11c and 11d describe one embodiment of the generation of this value. The table of Figure 11d illustrates the generation of the combined mask from the collection of output masks.

Compressed Stream Selection Logic

Figure 10 shows a block diagram of the selection logic 612/614/616 and the output stream generation logic 618. The compressed stream selection logic 612/614/616 collects the output counters and the output masks from each of the entries from the results calculation block 606, and generates indices and counts for the output stream generator 618, along with the Reset Value that is sent back to each entry. The indices point to the entries that generated the selected counts. The main function of the Selection Logic 612/614/616 is to find the largest blocks to be compressed out of the input stream, i.e., the largest contiguous match. This is accomplished by finding the largest output count from any entry. Because of the parallel compression, i.e., because a plurality of symbols is operated on in parallel, there could be multiple compressed blocks that need to be sent to the output. Because of this, in the four symbol parallel embodiment, two counts and three indices are provided to the output logic 618. These are referred to as the Previous Count and Index, the Max Count and Index, and the LZ12 index.

Selecting the largest count with a Mask of 11111 generates the Previous Count and Index. This indicates a compressed block that ended with the first data input of this cycle (i.e. the first data input or first symbol could not be compressed with this block). The Index is simply the entry number that contained the selected count. Selecting the largest count with a mask that is not 11111 generates the Max Count and Index. This indicates a compressed block that includes one or more of the 4 symbols received on this cycle. The mask from this entry is also forwarded to the output generator 618. The LZ12 index points to any block that returned a mask of 01111, which is the "special case". The special case includes a contiguous match of one or more middle symbols as described above. A

combined compress mask block 616 generates a combined compress mask comprising a logical AND of all of the masks, and forwards this to the Output Generator 618.

Finally, the selected Max Mask and the Reset Value column in the table of Figure 11 are used in generating a Reset Value. This reset value is distributed back to all entries, and the entries will reset their counters to the minimum of this value, or their present value.

Figure 12 - Output Stream Generator Flowchart

The output stream generator 618 logic (Figure 10) generates the output stream according to the flowchart shown in figure 12. The term "CCM" in this flowchart refers to the Combined Compress Mask, and CCM(0) is the least significant bit as used in the table of Figure 11. The output generator 618 sends out either uncompressed data, which includes the proper flags to indicate that it is not compressed, or a compressed block which includes a flag to indicate this is a compressed block, along with an encoded count and index that is used by the decompression logic to regenerate the original input.

As shown, in step 721 the method determines if previous count equals zero. If no, then the method sends out the compressed block in step 723 and adjusts the max count to 4 or less in step 725. Operation then advances to step 727. If previous count is determined to equal zero in step 721, then operation proceeds directly to step 727.

In step 727 the method determines if Max Cnt equals zero. If not, then the method determines in step 729 if Max Mask equals 10000. If not, then the method sends out the compressed block in step 731. Operation then advances to step 735. If Max Cnt is determined to equal zero in step 727 or if Max Mask is determined to equal 10000 in step 729, then operation proceeds directly to step 735.

In step 735 the method determines if CCM (3) equals zero. If not, then the method sends out data zero in step 733. Operation then advances to step 737. If CCM (3) is determined to equal zero in step 735, then operation proceeds directly to step 737.

In step 737 the method determines if CCM (4,2,1) equals 011. If not, then in step 739 the method determines if CCM (2) equals 1. If not, then in step 741 the method sends out data zero, and operation proceeds to step 745. If CCM (2) is determined to equal 1 in step 739, then operation proceeds directly to step 745. In step 745 the method determines if

CCM (1) equals 1. If not, then in step 747 the method sends out data zero. Operation then proceeds to step 749. If CCM (1) is determined to equal 1 in step 745, then operation proceeds directly to step 749.

If CCM (4,2,1) is determined to equal 011 in step 737, then in step 743, the method sends an LZ12 compressed block. Operation then proceeds to step 749.

In step 749 the method determines if CCM (0) equals 1. If not, then the method sends out data zero in step 751. Operation then completes. If CCM (0) is determined to equal 1 in step 749, then operation completes.

If single byte compression is being performed by this logic, i.e., if individual symbols are being compressed, additional indices for each of the byte matches should be generated by the Selection Logic to allow the Output Generator to compress these. Otherwise, the output generation logic should also handle the cases where outputs of a compressed stream result in a single byte non-compressed output and adjust the flags accordingly. Previous Data3 may also be required by the output generator 618 in the case that the previous match is a count of one. Preferably, one method of handling single byte matches would be to adjust the table of Figure 11 to not allow generation of single byte compare masks because single byte compares normally force the compressed stream to increase in size. For example, in the 10xx rows, if the saved count is 0, count out should be 0 along with a mask of 11xx to prevent the generation of a compressed block for the D0 single byte match.

Figure 13 - Parallel Algorithm Example

Figure 13 illustrates a parallel algorithm example. Assume a window (history table length) of 16 entries, that has been initialized to the following values: Entry 0 = F0, Entry 1 = F1 ... Entry 15 = FF. Also, assume that all of the entry counters are zero. The below sequence shows state changes for the four indicated inputs.

In state 0, the input data, in the order received, is F9, F8, F7, C0. The input data is shown in the arrival order from right to left in Figure 13, i.e., the input data D3:D0 = C0,F7,F8,F9. In state 0, the input finds a match of the first 3 symbols in entry 9. This results in those three symbols being replaced in the output stream by compressed data

indicating a count of three and an index of 9. The output mask value "18" prevents these uncompressed symbols from being included in the output stream, since the compressed data is being output to represent these symbols. Also in state 0, the symbol C5 is determined to not match any entry in the history table. Thus, the symbol C5 is provided in the output stream in uncompressed form. Thus the output in state 0, from right to left, is: C0, (9,3).

In state 1, the input data, in the order received, is B5, F2, F1, F0. The symbol B5 does not match any entry in the history table. Thus, the symbol B5 is provided in the output stream in uncompressed form. In addition, in state one, three input symbols match 3 symbols in entry 7. Note that the matches are in previous entries, but the results calculation for this match occurs in entry 7. In other words, the actual matching entries are entries 6, 5, and 4. However, this match is detected by entry 7, since entry 7 compares the 4 input symbols with entries 7, 6, 5, and 4. Compressed data is not generated for this match in state 1 because the entry does not know if the match will continue with the next set of input symbols, and thus the output count is zero. The mask value for entry 7 prevents the matching data from being included in the output stream. Thus, the output in state 1 is B5. The count value for entry 7 is updated to three, as shown in state 2, to indicate the 3 matches in state 1.

In state 2, the input data, in the order received, is F9, F8, F7, B5. The matching in entry 7 continues for 3 more symbols, and then ends. Thus, entry 7 outputs a count of six and a mask for the new matching symbols. In addition, entry 6 matches with the symbol B5. Thus, entry 6 updates its count to one in state 3. However, since symbol B5 is the last symbol in this group of input symbols, the entry does not know if the match will continue with the next set of input symbols. Thus, for entry 6 the output count is 0 and the mask value will prevent that symbol from being output. Thus the output in state 2 is (7,6)

In state 3, no further contiguous matches exist for the symbol B5 from state 2. Thus, for entry 6, the output count is 1 from entry 6 for the B5 input after stage 2. In addition, no match is detected for input symbol E2, and thus E2 is output as an uncompressed symbol. In state 3, a match is detected with respect to the middle symbols C0 and B5. This match comprising solely middle symbols is detected by entry 9, and thus the 0F Mask is output from entry 9. This mask is the special case mask that indicates the

two symbols centered in the input (B5C0 in this example) can be compressed out. The actual compressed output data or block will include a flag, a count of 2 and the index 9. Thus the output from state 3, from right to left, is (9,2), E2, (6,1). In an embodiment where individual symbols are not compressed, the output is (9,2), E2, B5, as shown in the alternate output box.

The final state in this example, state 4, has a 1 in the count for entry 7 as a result of a match of F3 with entry 4 in state 3. The mask from this match prevented the sending of the F3 to the output stream in state 3. If this were the end of the input stream, the window is flushed, resulting in the single symbol compression block for this match. The output would show a match of 1 at index 7. Thus, if the input in state 3 is the final data received, then the final output for the stream is (7,1). Alternately, the single symbol match could be sent uncompressed as symbol F3, as shown in the alternate output box.

Compare Logic

The compare logic 612 and 614 (Figure 10) in stage three, which is used to find the largest count may be specially designed to be able to complete in one cycle. The counts are especially critical because stage 2 must first choose to send 0, count, count+1, count+2 or count+3. The count from all entries is then compared to find the largest.

As shown in Figure 14, straightforward greater-than compare of two multi-bit numbers requires three levels plus a selector. If the number is six bits, this compare will require around 30 gates, and the selector will require an additional 18 for the selector for 48 gates per 2-way compare. A stacked compare (64 to 32, 32 to 16, 16 to 8, 8 to 4, 4 to 2, 2 to 1) would require 6*5 levels of logic, and 48*63~3Kgates.

With standard .25um process technology, the time through the compare should be about 1.25nS (.25ns per XOR, .5ns 6wayAnd/Or). The selector would take an additional .3nS for 1.55nS per compare. This stacked compare would then require 1.55nS*6 = 9.3nS. This doesn't include the selection and distribution of these counts from the source. For operation above 100Mhz clocking, the timing is too limiting for proper operation.

In order to increase the speed, a novel four way parallel compare can be used, as shown in Figure 15. This embodiment only requires 3 levels of compares (64 to 16, 16 to 4,

4 to 1), however, more two-way compares are required (6 per 4 way compare) and an additional And/Or is required before the selector. This design would then require 126 compares and 21 selectors for 126*30+21*33~4.5Kgates. But the resulting delay would be (1.55+.3ns)*3 Levels = 5.55nS. This timing allows for high-speed parallel compression of the input data stream. The table of Figure 16 describes the Select Generation Logic.

Lossy Compression Algorithm

As indicated in US patent disclosure entitled "Memory Controller Including Embedded Data Compression and Decompression Engines", filed June 5, 1995, serial number 08/463,106, whose inventor is Thomas A. Dye, it is also desirable to implement some of the compression formats as "lossy". The term "Lossy" implies a compression/decompression operation where data is altered and is represented by an approximation of the original data after decompression.

Referring to Figure 21, some compression conversion formats preferably use lossy compression while others use lossless compression. In the preferred embodiment, texture 302, image data (Compressed block 380), video data (Compressed Block 380), and display data 300, and in some cases "Z" or depth data, are compressed with the lossy algorithm. Alternate embodiments include any of these formats or additional formats to be compressed with the lossless compression algorithm. Control data, programs, VDRL, or 3D parameter data, or any other data required to be decompressed without loss from the original content is compressed using the lossless parallel compression process according to the present invention.

Figure 17 – Lossy Compression and Decompression Engines

Figure 17 illustrates the preferred embodiment of the lossy compression engine 575 and the lossy decompression engine 555. These two engines preferably are located within the parallel compression and decompression unit 251.

The lossy compression engine 575 and the lossy decompression engine 555 may be separate blocks or integrated as a single unit. The engines 575 and 555 may be implemented in any of various manners, including discrete logic, a programmable CPU,

DSP, or microcontroller, or reconfigurable logic such as an FPGA, among others. Preferably, the lossy compression engine 575 performs the lossy compression algorithm for image, texture, video, and depth data.

Data in either RGB or YUV color format is presented to the lossy compression engine 575 by the switch logic 261 of the memory controller 220. If such data is in the RGB format, a source converter 762 is used to encode the RGB to a luminance (Y) value (encoded to YRB). This conversion process operation is standard for those who are knowledgeable in the art. The reason for this conversion is to improve color replication across the compression and subsequent decompression procedure. Note that the YUV data is not converted by block 762, but rather is treated by the compression algorithm the same as the YRB data previously converted by the source converter 762.

The data is selected by mux 764 for storage as normal data by SRAM store 770 and for min & max calculation by 768 and 766 respectively as described further. The data that resides in SRAM store 770 is selected for values according to the tables of Figures 18 and 19. The YRB/YUV values are interpolated by select switch 772 under the control signals generated by control logic located within the Max Y 766 and Min Y 768 units. The lossy data encoder 774 performs the control bit insertion into the selected values that are output by the YRB select switch 772. Lossy compressed data from the lossy compression Engine 575 is output to the memory interface logic 221 for storage in the main system memory 110.

Likewise the lossy decompression engine 555 receives the compressed data from the memory interface logic 221 to perform the lossy decompression operation. Data is first processed by the compressed stream separator 776 which strips off the header for process control information and sends appropriate signals to the lossy data decoder 778 and the pixel replicate logic 780. The lossy data decoder 778 controls the replication process performed in the pixel replicate unit 780. Data Min and Max Y values with the associated Red and Blue (or U and V) can be positioned back preferably into a 4x4 array of output pixels. The final step performed by the Y to G converter 782 is to convert the YRB/YUV data format back to the original RGB format as indicated by the header that accompanied the block of compressed data. For decompression of YUV data, the Y to G conversion process is skipped and the data is output directly from the Y to G converter 782. In

alternate embodiments other color source formats can be used, as the compression method operates with a luminance value to determine the minimum and maximum intensity within the group or block of data under compression.

In the preferred embodiment the lossy compression algorithm starts with a 4x4 block of pixels in RGB format and compresses them to various size blocks depending on the attributes of that 4x4 block. Alternate embodiments may use other initial source data block sizes with simple extension to the following process. Also in the preferred embodiment each block could be encoded to a different size, and its size is encoded with the data so the decompression engine can function properly. Alternatively, some applications such as consumer appliances and embedded DRAM require a "fixed" compression ratio in order to accommodate a fixed size memory environment. Fixed compression ratio allows the software to allocate memory in a known size and also compensates for overflow of data past the physical limit of the memory size. In this alternate embodiment, where a fixed compression ratio is required, the lossy algorithm is easily changed to eliminate special cases, which in the preferred embodiment allow a better compression ratio.

Also, in an alternate embodiment the CPU 102 may perform the compression and/or decompression in software according to the present invention. In another embodiment, the decompression process can be performed by logic while the compression can be performed by software executing on the CPU 102.

Data input may originate in the YUV format (typically video) or the RGB format (typically graphics) and may also be combined with alpha for transparency effect. In the preferred embodiment, if the data to be compressed is in Red, Green and Blue format, data is converted to the proper data format of Y (luminance), Red and Blue or is left in YUV format if that is the original source format. During the source read process the data format is converted to the preferred format and a number of compare steps are performed on each block as indicated. The Y values of the block of 4x4 pixels during load are compared to the previous values for the maximum and minimum Y values of two pixels. Once found the associated R and G values are stored corresponding to such minimum and maximum Y values. Thus the maximum Y and minimum Y are determined for each block. As the data

for each pixel is read the maximum and minimum Y are located, the associated R, B and Alpha values for the minimum and maximum Y pixels are also stored 770.

For compression operation without alpha components, Figure 18 indicates the algorithm used to output a block. Likewise, for the lossy compression operation with alpha, values in Figure 19 are used. Now with reference to the tables of Figures 18 and 19, P bits accompany the compressed data such that during the decompression stage output pixel locations can be determined. If 16 P bits are required, then each pixel is compared with the two colors found in the block, and a 0 indicates that pixel is the Min color (Y_{min}, R_{min}, B_{min}, A_{min}) or a 1 indicates that pixel is the Max color. When greater than two colors or alphas are present as determined by minimum 768 and maximum 766 Y logic, 32 bits are used. When 32 P bits are used the compression unit calculates intermediate Y values at $1/6^{th}$, ½, and $5/6^{th}$ between the Max and Min Y values. The Y value of each pixel is then compared with these values, and if less than or equal to the $1/6^{th}$ value, 00 is used for this pixel. If greater than the $1/6^{th}$ value, but less than or equal to the ½ value, a 01 is used for this pixel. Likewise, for 10 (between ½ value and $5/6^{th}$ value) and 11 (greater than $5/6^{th}$ value). The decompression engine will calculate the $1/3^{rd}$ and $2/3^{rd}$ values between Y_{max} and Y_{min}, and if the value for the pixel is 00, Y_{min} will be used. If 01, the $1/3^{rd}$ value is used, 10 uses the $2/3^{rd}$ value, and 11 uses the Y_{max} value. During the decompression process, the Y, R, B color format is reconverted into the original data format R, G, B, or Y, U, V. For application or system requirements where a fixed compression ratio is required, the default algorithm can use the last entries referenced in Figures 18 and 19 for each 16 and 32 bit data input formats. Alternate embodiments could use a larger or fewer bits for each pixel's P bits, or P bits based on individual colors for the pixel. In addition, alternate embodiments and variations of the lossy compression may yield less compression but higher image quality and fixed compression ratios.

Figure 20 - Combined Compression

Due to the nature of the compression requirements the preferred embodiment introduces a new method to achieve high quality fixed or variable image and video compression ratios using a combination of both the lossy and lossless engines. The IMC

140 compresses multiple data types and formats as discussed previously in this disclosure. When image data is compressed with only a lossy algorithm, image data with high detail can be blurred or washed out. Prior art performs lossy compression on image data with discrete cosine transforms by conversion into the frequency domain. These practices are expensive due to the high bandwidth requirements of the real time transformation for video and graphics from the time domain to the frequency domain.

In order to solve these issues, a combination of both lossy and lossless engines 575 and 570 running in parallel is performed, and outputs from one of the engines is selected based on a criteria.

As shown in Figure 20, the original source data 120, e.g., from disk, subsystem, or CPU 102, is transmitted into the input switch 261 across the input bus, where the bus may be an embedded local data or CPU bus or be a proprietary internal design bus. The input switch 261 performs the determination of address and qualification for block size and compression operation. The data then is sent to both the parallel lossless compression engine 570 and the lossy compression engine 575, which performs the proper compression before storing into the SRAM store memory 581 and 582, respectively.

The source data is thus read into both the parallel lossless compression engine 570 and the lossy compression engine 575 in parallel. Both engines compress data of equivalent input block sizes, while compressed output sizes from each engine may vary.

In the preferred embodiment of Figure 20, an error term determines the selection of either the lossy or the lossless compression results for insertion into the compressed stream. The lossy compression engine 575 may generate the error term during the compression of the incoming data stream. More specifically, an array compare unit 584 generates the error signal in response to output from the lossy compression engine 575. The error signal is preferably generated based on difference between the Min Y and Max Y values. Alternatively, during the lossy compression process, the original data is subtracted from the encoded or lossy compressed data to produce the error term. This error then determines if the block to insert in the compressed stream is either lossy compressed or lossless compressed form. The error signal is provided to an output format switch or multiplexer 586, which selects the compressed data from either the lossless engine 570 or the lossy

engine 575. As shown, the outputs of the lossless engine 570 and the lossy engine 575 are temporarily stored in SRAM stores 581 and 582 prior to being provided to the output format switch 586. If the error signal is below a certain threshold, indicating a low error in the compression output of the lossy compression engine 575, then the output of the lossy compression engine 575 is used. If the error signal is above the threshold, then the error in the compressed output from the lossy engine is deemed unacceptably high, and the output from the lossless engine 570 is selected.

Thus, for areas that show a high error due to the magnitude of the difference in luminance, the lossless parallel compression data is used. For data that shows a minimal threshold of error, the lossy compressed data is used. The advantage of this technique is that blocks of image to be compressed with noise will compress better with the lossy engine. Likewise, blocks that have repetitive detail, high frequency imagery or detailed repetitive data will compress more effectively with the lossless parallel compression.

During the write of compressed blocks, the header includes a tag bit used as an indication of the type of compression used. This tag bit is used during decompression to apply the proper decompression procedure to the data.

The error term selection can also be a dynamic function to assure a fixed compression ratio. In this embodiment, if a fixed compression ratio is desired, the dynamic threshold can be adjusted to vary the magnitude of the error deemed acceptable for lossy compression. A running tally of the current compression ratio is used to dynamically adjust the threshold value, which determines where the lossless compression blocks are used instead of the lossy compressed blocks. This operates to degrade the image, if necessary, by selection of additional lossy compression blocks in lieu of lossless compression blocks. If the run rate of the current block is at the required compression ratio, then the threshold is set to the default value. If the current run rate is over-allocated, the error threshold value will increase such that output selection is from the lossy compression engine 575. Thus, a dynamic compression error threshold determines how to adjust the ratio of lossy to lossless data in order to achieve a guaranteed compression ratio.

During decompression, preferably the output format switch 588 first strips the header for determination of decompression engine output selection. In one embodiment, the

compressed data is decompressed in parallel by both engines 555 and 550. In this embodiment, during decompression, the header of each block determines, preferably after completion of the decompression operation, whether the destination pixel is selected from the lossy decompression engine 555 or the lossless decompression engine 550. The output format switch 588 performs the selection of decompression engine output.

In another embodiment, only the selected decompression engine, either 555 or 550, is applied to the data. In this embodiment, the compressed data is efficiently allocated to the proper decompression engine, depending on the mode of compression as determined by the header.

Figure 21 - Compression Formats

As shown in Figure 21, the preferred embodiment of the present invention allows faster memory access time using a plurality of compressed storage formats. The system may be designed to optimize the compression and decompression ratios based on the type of system data. Data that is used for programs or used to control the processing of other data is compressed and stored in a lossless format (lossless compression). Likewise, data that can be compressed with loss during recovery or de-compression is compressed in a lossy format. Thus, each format has a specific address and memory orientation for best decompression rate and storage size. In addition, each specific compression and decompression format scales in bandwidth performance based on the amount of cache memory used to store uncompressed memory during the compression and decompression process.

Referring to Figure 21, in addition to the lossless format and lossy formats, the IMC 140 preferably contains further multiple compression and decompression formats for efficiency and optimization of bandwidth within the memory controller device. Data Source blocks 310, 320, 330, 340, and 350 represent the compression format of data that is read from system memory 110, written from the CPU 102, read from the non-volatile memory 120, read from the I/O system controller 116, or read from the internal graphics blocks within the IMC 140 device, or alternatively as in prior art Figure 1, read from the PCI or AGP buses 107 to the IMC 140. Destination blocks 360, 370, 380, 390, 396, 300

represent the compression format of data that is written to system memory 110, or read by the CPU 102 (transferred to the CPU 102 in response to a CPU read), written to the non-volatile memory 120, written to the I/O system controller 116, written to internal graphics blocks within the IMC 140 device, or alternatively as in prior art Figure 1, written to the PCI or AGP buses 107 from the IMC 140. Therefore, blocks 310, 320, 330, 340, 350 are considered the data source formats where data flows into or is generated within the IMC. Blocks 360, 370, 380, 390, 396, and 300 are destination formats where data flows out of the IMC. It is noted that destination formats become source formats on subsequent accesses by the IMC 140. Thus a compression format may be referred to as source format / destination format.

Blocks 302, 304, 306, 308 and 309 represent the data type of the data. These data types include texture data 302, 3D-DL 304, 2D-DL 306, DV-DL 308 and VDRL 309. These data types are discussed briefly below.

VDRL, Indirect Compressed Lines

One form of data in the preferred embodiment is video display refresh list (VDRL) data as described in U.S. Patent Number 5,838,334, referenced above. VDRL data comprises commands and/or data for referencing pixel/video data on a span line basis, typically from various non-contiguous memory areas, for refresh of the display. VDRL compressed data is expected to be a long stream of start and stop pointers including various slopes and integer data. Such data is compressed with the lossless compression and decompression process in the preferred embodiment. The following VDRL context register fields in the graphics engine can be programmed to cause screen data to be written back to system memory as lossless compressed screen lines 390(or sub-lines) during VDRL execution:

```
DestEn
DestType = {Linear, XY, or LineCompressed}
pDestTopLinePtr              // Pointer to compressed pointer list
pDestTopLine          // Pointer to screen data
DestMode = {Draw&Refresh | DrawOnly}
```

DestPixFmt

DestPitch

When enabled, each screen line (or span line) that is rendered or displayed (based on processing one or more VDRL segments) is compressed independently (for each screen line, a new compression stream is started and closed) and written back to memory at the current byte offset into pDestTopLine. In addition, the graphics engine writes back a pointer to the compressed screen line at the current pointer offset into pDestTopLinePtr. The current offsets into pDestTopLine and pDestTopLinePtr are managed by the graphics engine. The compressed screen data 300 and corresponding pointer list can be referenced as a compressed window by a subsequent VDRL 309. Preferably the workspace associated with the compressed window includes the following fields used by the graphics engine to indirectly access the compressed screen data:

pTopLine

pTopLinePtr

SrcType = {Linear | XY | LineCompressed}

PixFmt

Pitch

Since screen lines are compressed on a line 390 (or sub-line) basis, the subsequent VDRL 309 only has to reference those lines that are needed for the current screen being refreshed.

Note: 3D-DL 304 and DV-DL 308 can also render indirect compressed screen lines 396 in a similar manner. However, the resulting indirect compressed screen lines are to be consumed by subsequent VDRL 309.

Note: DV-DL 308 is fundamentally based on processing and drawing blocks. For implementations that do not have enough storage blocks to cover the width of the screen being drawn, screen lines 390, 300 are compressed back to memory on a sub-line basis.

Static Data

For each independent triangle, the 3D-triangle setup engine generates two lossless compressed static data blocks using standard linear compression 360: an execution static data block, and a graphics engine static data block. For a given 3D window or object, all static data is written starting at a particular base address (pTopStatic). Each static data block is compressed independently (for each static data block, a new compression stream is started and closed) and written back to memory at the current compressed block offset into pTopStatic. In addition, the 3D triangle setup engine writes back a pointer to the compressed static data block (pStatic) in the appropriate static pointer line bucket. The format of pStatic comprises the following fields: static data block pointer offset, static format (indicating whether the data is compressed or not), the number of compressed blocks associated with the execution static data block, and the number of compressed blocks associated with the graphics engine static data block. Note that the number of compressed blocks for each static data block type is used to instruct the decompression engine 550 how much data to decompress.

3D-DL

A 3D-DL comprises a 3-dimensional draw list for rendering a 3-D image into memory, or onto the display. For each 3D window line (or sub-line), the 3D execution engine generates a lossless compressed stream of a 3D-DL 304. Each 3D-DL line is compressed independently (i.e. for each 3DDL line, a new compression stream is started and closed) and the resulting compressed 3D-DL line 390 is written back to memory 110. It is not necessary for consecutive lines of 3D-DL to be contiguous in memory. In addition, the 3D execution engine of the IMC 140 may write back a 3D-DL pointer to the compressed 3D-DL line 390 at the current pointer offset into the 3D-DL pointer list (p3DDLPtr). The resulting compressed 3D-DL lines 390 and corresponding 3D-DL pointer list 304 is parsed and consumed by the 3D graphics engine 212. The graphics engine 212 uses the following 3D-DL context register fields:

p3DDL

p3DDLPtr

The context register fields operate to provide context information to the IMC 140 during execution of a 3D-DL.

Note: Since 3D-DL is compressed on a line 390 (or sub-line) basis, only the visible portion of a 3D window (based on feedback from VDRL window priority resolution) may need to be drawn.

Textures

Texture data 302 for 3D rendering is also compressed and decompression according to the present invention. The lossy algorithm preferably compresses images. In an alternate embodiment, the parallel combination of lossy and lossless algorithms can be used for improved image and texture map quality without added time delay. Texture data 302 is typically compressed and decompressed in a block compression format 380 of the present invention. The logical format of a lossy (or lossless) compressed texture table for a given scene with T textures, is as follows:

```
pTopTex ->
opTex0 ->
pLod0Blk0 ->            8x8 compressed texture sub-blocks
pLod0Blk(last) ->
pLod(last)Blk(last) ->
opTex1 ->
pLod0Blk0 ->
opTex(T-1) -> . . .
```

pTopTex is the base pointer to a compressed texture table. pTopTex is loaded into the graphics engine 212 on a per 3D window basis. opTex is an offset into pTopTex that provides the graphics engine 212 with a pointer to the first compressed texture sub-block (i.e., LOD0, sub-block 0) associated with the targeted texture. opTex is a field located in a group attribute data block, RenderState. RenderState contains attributes shared by groups of triangles. The group attribute data block pointer, pRenderState, is contained in each 3D-DL

304 segment. Using pTopTex, opTex, and all of the texture attributes and modifiers, one of the graphics engine's texture address generation engines determine which critical texture sub-blocks 380 (pLodBlk) to prefetch.

The size of a texture sub-block 380 in the preferred embodiment will be 8x8 texels. The compressed texture sub-blocks are read into the compressed texture cache. Note that the pLodBlk pointers point to 8x8 compressed texture sub-blocks 380.

DV-DL Video

The DV-DL format comprises a digital video draw list for rendering digital video into memory or onto the display. The block compression format 380 can also be used for video and video motion estimation data. In addition, Display data 300 is also preferably stored in compressed format according to the present invention. The display data 300 is expected to be sequentially accessed RGB or YUV data in scan line blocks typically greater than 2K bytes. The preferred method for compression of display data 300 is to line compress 390 the entire span line, preferably in the parallel lossless format.

Video input data is also compressed preferably in any of the formats, lossless, lossy, or a combination of lossy and lossless according to the present invention. Video data is typically and preferably compressed and decompressed in two-dimensional blocks 380 addressed in linear or X/Y format.

Each data type has a unique addressing scheme to fit the most effective natural data format of the incoming source format.

For special graphics, video, and audio data types 306, 308 and 310 the data types can be associated with a respective compression format to achieve optimal compression ratios for the system.

Blocks 310 and 360 represent a lossless or lossy compression and decompression format of linear addressed compressed or decompressed data blocks as specified by the CPU 102 and system software. Data block size and data compression types are dependent on the bandwidth and cost requirements of the application and system respectively. Source data applied to block 310, if coming from the system memory, will be decompressed and written to the destination as normal (uncompressed) data or data which has some loss

associated with the decompression process. The input bandwidth of compressed data provided to block 310 is equal to the bandwidth required by normal non-compressed data divided by the difference of the compression ratio. The compression ratio is a function of multiple constraints, including compression block size, data type, and data format. Further, the bandwidth of the uncompressed destination data is equal to the original uncompressed source data bandwidth. In addition, source data can be uncompressed "normal" data that is compressed and written to the destination in one of many compression formats as indicated by blocks 360, 380, 390, and 396.

Source data block 320 represents incoming data that has not been altered by compression. In this case data which represents a texture type can be written in the compressed block format 380 for optimal use of 3D texture memory space. Likewise, 3D-Draw (3D-DDL) type data can be received as source data in an uncompressed format 320 and can be processed and formatted for output in either uncompressed 370 or line compressed 390 destination formats. Similar operation can occur when the source is already in Compressed block format 330.

Compressed line 340/390 for example may be generated from VDRL 309 instructions and stored in partial compressed line segments 340/390 for later usage by another requesting agent. These compressed line segments are addressed in standard linear addressing format.

Intermediate compressed line segments 350/396 are special cases of conversion from compressed blocks 330/380 to compressed intermediate lines 350/396. Compressed intermediate lines are used as a conversion technique between compressed block 330/380 and the digital video draw list (DV-DL) 308.

Display data 300 can also be compressed and is typically compressed in a lossless format that is linear complete span lines. During the refresh of video to the display, the display compressed span lines 300 which have not been modified by the 3D graphics engine 212 are decompressed for display on the respective display device span line.

Video and Texture data 302, for example, are preferably in uncompressed 320/370 or compressed block 330/380 formats. Block formats 330/380 are typically 8x8 blocks that have representation of X/Y address but are referenced in system memory as linear 64 bytes

with a pitch of 8bytes. In the compressed block format 330/380, decompression results in 32x32 texture blocks also addressed in X/Y format.

Instruction lists, such as VDRL (video display refresh list) 309, DV-DL (digital video draw list 308, 3D-DL (3-D draw list) 304 preferably are stored in a lossless compressed format with linear addressing. CPU data is also preferably stored in a lossless compressed format with linear addressing. These instruction lists are executable to render pixel data into memory in response to geometry lists or to access video/pixel data from memory for display on the display device. The draw results of these also have formats as indicated in Figure 21. For example, uncompressed linear addressed data 320 as a source may be manipulated and read by the 3D-DL 304 instruction list, and stored compressed in compressed line 390 format or Uncompressed 370 data format. Each operator indicated in Figure 21 has a preferred format for data transition and storage.

Data which is type 2D-Draw list 306 is received as source data in uncompressed 320 format or block compressed 330 format. For 2D-DL data type 306, the output data can be in uncompressed 370 or Intermediate line compressed 396 formats.

For digital video draw lists (DV-DL) 308, the source data of the DV-DL 308 is received in uncompressed 320 format or block compressed 330 format which is output to the destination in intermediate line compressed 396 format.

Source data of the VDRL data type is received in either uncompressed 320, Compressed line 340, or intermediate compressed line 350 formats, and is output to the destination address as compressed line 390 or directly to the display device 300.

Lastly, data of the Display format type 300 is typically normal or lossless compressed with a linear span addressing format.

As indicated in US Patent Number 5,838,334, "workspace areas" are located in memory to define the windows or object types. In one embodiment, the relationship between such workspace regions and the compression and decompression operation of the present invention is as follows. Each "workspace" contains a data area which indicates the compression type and quality (if lossy compression) for reproduction of the window or object on the display. The Application Software (API), Graphical User Interface (GUI) software or Operating System (OS) software can determine the type and memory allocation

requirements and procedures to optimize the cost, performance and efficiency of the present invention. Windows or objects that have been altered from the original content or that have been resized can be represented with a plurality of quality levels for final representation on the display as indicated in the window workspace areas of the main system memory. In addition, 3D objects or textures can contain the compression quality attributes as well. Thus, by assignment of compression type, address format, and quality of representation in the individual window or object workspace area, the system can be optimized for cost and performance by the elimination of memory size and bandwidth requirements.

Data types texture data 302, 3D-draw lists 304, 2D-draw lists 306, Digital video draw lists 308, and Virtual (video) Display Refresh List 309 all represent the audio, video and graphics media formats of the IMC as referenced in U.S. patent number 5,838,334.

The core compression block formats allow multiple data types from various sources as inputs. The compression and decompression formats attempt to compress the data into the smallest possible storage units for highest efficiency, dependent upon the data type of the data received. To achieve this, the memory controller 210 understands the data types that it may receive.

Therefore, the IMC 140 of the present invention reduces the amount of data required to be moved within the system by specific formats designed for CPU 102, Disk 120, system memory 110, and video display, thus reducing the overall cost while improving the performance of the computer system. According to the present invention, the CPU 102 spends much less time moving data between the various subsystems. This frees up the CPU 102 and allows the CPU 102 greater time to work on the application program.

As discussed further below, data from the CPU may be compressed and stored in linear address memory with variable block sizes. This data from the CPU may be unrelated to the graphics data, and may result from invalidation of cache lines or least recently used pages (LRU), or requested memory from a CPU-based application. In this embodiment the driver requesting compression will handle the memory allocation and directory function for both the compressed and uncompressed data.

Latency and Efficiency

The memory Controller 220 minimizes latency of read operations by a plurality of novel methods. Each method is discussed further in reference to the preferred embodiment. Most of the control functions for latency reduction are located in the switch logic 261, and further located in the compression switch logic 516, the decompression switch 512 and the normal memory switch 514. Locality of data addresses to compression blocks and L3 data cache blocks also play a major role in latency reduction. The various latency reduction and efficiency methods include: Parallel compression/decompression (described above); Selectable compression modes; Priority compression mode; Variable compression block size; the L3 Data Cache; and Compression Reordering.

Figures 22 and 23 - Selection of Compression/Decompression Mode Based on Criteria

The parallel compression and decompression unit 251 can selectively perform a compression / decompression mode or type (compression mode) based on one or more of: requesting agent, address range, or data type and format, again as indicated in US Patent application Serial No. 08/463,106. Examples of the compression / decompression modes (compression modes) include lossless compression, lossy compression, no compression, and the various compression formats shown in Figure 21. The compression modes may also include varying levels of lossy compression for video/graphical objects or windows which are displayed on the display. Thus the IMC 140 can selectively perform lossless compression for first data, lossy compression for second data, and no compression for third data.

Figures 22 and 23 are flowcharts illustrating selective use of compression and decompression schemes. The method of Figures 22 and 23 is preferably performed by the memory controller comprising the compression/decompression engine. The memory controller is preferably a system memory controller for controlling system memory, wherein the system memory stores application code and data executed by the CPU.

As shown, the method in step 802 first receives uncompressed data. The data may be CPU application data, operating system data, graphics/video data, or other types of data. The data may originate from any of the various requesting agents.

In step 804 the method determines a compression mode for the data. The compression mode preferably comprises one of lossless compression, lossy compression, or no compression. Other compression modes include either the lossless or lossy types above in combination with one of the compression types shown in Figure 21, e.g., either compressed linear, compressed block, compressed line, or I-compressed line.

The compression mode is preferably determined in response to one or more of: an address range where the data is to be stored; a requesting agent which provides the data; and/or a data type of the data.

Where the address range is used to determine the compression mode, the method analyzes the destination address received with the data to determine the compression mode, wherein the destination addresses indicating a storage destination for the data in the memory. For example, assume a first address range is designated with a lossless compression format, a second address range is designated with a lossy compression format, and a third address range is designated with a no compression format. In this case, step 804 of determining the compression mode comprises analyzing the destination address(es) to determine if the address(es) reside in the first address range, the second address range, or the third address range.

Where the requesting agent is used to determine the compression mode, the method determines who is the requesting agent and then determines the compression mode based on the requesting agent. For example, if the requesting agent is a CPU application or associated driver, then a lossless compression should be applied. If the requesting agent is a video/graphics driver, then lossy compression may be applied.

Where the data type is used to determine the compression mode, the method examines the data type of the data and determines the compression mode based on the data type of the data. Using the example above, if the data comprises application data, the compression mode is determined to be lossless compression. If the data comprises video/graphics data, then the compression mode may be lossy compression. In the preferred embodiment, the determination of the compression mode is preferably inherently based on data type of the data, and the use of address range or requesting agent in determining

compression mode may be implicitly based on the data type being stored in the address range or originating from the requesting agent.

Further, the compression modes may comprise varying levels of lossy compression for video/graphical objects or windows which are displayed on the display. Thus a lossy compression with a greater compression ratio may be applied for objects which are in the background of the display, whereas lossy compression with a lesser compression ratio may be applied for objects which are in the foreground of the display. As noted above, for graphical/image data, in step 804 the compression mode may be determined on a per-object basis, e.g., based on whether the object is in the foreground or background, or based on an attribute of the graphical object. For example, 2, 4, 8, or 16 varying levels of lossy compression may be applied to graphical/image data, depending on attributes of the object.

In step 806 the method selectively compresses the uncompressed data based on or in response to the compression mode for the data. In step 806, the data is compressed using a lossless compression format if the compression mode indicates lossless compression for the data, the data is compressed using a lossy compression format if the compression mode indicates lossy compression for the data, and the data is not compressed if the compression mode indicates no compression for the data.

In step 808 the method stores the data in the memory. In step 808, the data is stored in the memory in a lossless compression format if the compression mode indicates lossless compression for the data, the data is stored in the memory in a lossy compression format if the compression mode indicates lossy compression for the data, and the data is stored in the memory in an uncompressed format if the compression mode indicates no compression for the data.

In the preferred embodiment, storing the data in the memory includes storing compression mode information in the memory with the data. The compression mode information indicates a decompression procedure for decompression of the compressed data. The compression mode information is stored in a non-compressed format regardless of the compression mode of the data.

The compression mode information is preferably embedded in the data, i.e., is not stored in a separate table or directory. In the preferred embodiment, a header is created

which includes compression mode information indicating the compression mode of the first data. As described below, the header is also used to store other information, such as an overflow indicator and overflow information. The header is preferably located at the top of the data, i.e., is stored at the beginning address, followed by the data, but may also be located at the bottom of the data or at designated points in the data.

In an alternate embodiment, the IMC 140 reserves space for an overflow tag and overflow table entry number in memory within the IMC 140. Thus, in this embodiment, the IMC 140 includes a separate overflow cache, entry table and control logic. In an alternate embodiment, the overflow indication can be processed by the same control and translation cache logic blocks used for a normal compression operation.

Referring now to Figure 23, decompression of the stored data is shown. In step 812 the method receives a request for the data.

In step 814 the method accesses the data from the memory in response to the request.

In step 816 the method determines a compression mode for the data in response to receiving the request. In the preferred embodiment, the compression mode is comprised in the stored data, preferably within a header comprised within the stored data. Thus the data is first accessed in step 814 before the compression mode is determined in step 816.

In step 818 the method selectively decompresses the data. The type or mode of decompression is selected based on the compression mode for the data. In the selective decompression of step 818, the data is decompressed using lossless decompression if the compression mode indicates lossless compression for the data, the data is decompressed using lossy decompression if the compression mode indicates lossy compression for the data, and the data is not decompressed if the compression mode indicates no compression for the data.

In step 820, after decompression, the method provides the data in response to the request.

Thus, to further reduce latency, certain selected data can be stored/retrieved with normal operation using no compression or with a selected compression mode such as lossless or lossy. This is preferably accomplished by address range comparison for Memory

management unit (MMU) blocks that contain special flags for "no-compression" indication. It is assumed that for power-on configuration, these non-compression address ranges may be set to the supervisor mode code and data blocks used by the operating system.

The MMU in the memory controller 210 can determine (e.g., 4096 byte range) what form of compression, if any, is used. In the preferred embodiment, this determination is based on compression fields located within the MMU translation table on a memory page boundary. In alternate embodiments, the compression type flags may be located on a plurality of boundary ranges. The method of using address range look-up to determine memory compression data types is further documented in patent disclosure titled "Memory Controller Including Embedded Data Compression and Decompression Engines", filed June 5, 1995, serial number 08/463,106, whose inventor is Thomas A. Dye.

Memory Allocation for Compressed Data - Priority and Normal Compression Modes

1. Priority Mode Compression

The IMC 140 includes two different compression modes for fast and efficient memory allocation and data retrieval. These two modes are referred to as "priority compression mode" and "normal compression mode". The "priority mode" architecture is a non-intrusive memory allocation scheme. Priority mode provides the ability to incorporate the MemoryF/X Technology, including the compression/decompression capabilities, for faster effective bandwidth, without requiring operating system software changes. In this case (without OS changes) the memory controller 210 of the IMC 140 is more tailored to bandwidth improvements than to memory size conservation. The compression and decompression operations increase the effective bandwidth of the system. The memory allocation and compression operations uses the additional memory freed up by the compression algorithm for the overflow space. The overflow space is used in cases where the lossless compression results in more data than the original data size before compression. The "priority mode" feature is used for systems that require faster data transfers and have no need for memory conservation.

In the case of priority mode operation, the overflow addresses are assumed to be in memory blocks previously reduced by the compression operation. Thus in priority mode

system software reallocation is not required to compensate for memory allocation and size. Any second level overflow or overflow that does not fit into the allocated overflow area provided by the memory allocation of the present invention is handled by a system level driver interrupt. In such cases where a real time event can not handle the second level interrupt delay, a fixed compression ratio of a required size can be used under the alternate embodiment previously disclosed.

The priority mode is used for compressing data and storing the compressed data in a memory in a computer system, wherein portions of the computer system are not required to account for the compression. In the priority mode method, the computer system, e.g., the operating system, first allocates a memory block for uncompressed data. The memory block is allocated on the assumption that the data stored there will be uncompressed data. The operating system is not required to account for the compression operation and may be unaware of the compression operation.

The memory controller may later receive uncompressed data and one or more corresponding destination addresses indicating a storage destination of the first data in the allocated memory block. In response, the memory controller compresses the uncompressed data to produce compressed data. The memory controller then stores the compressed first data in the allocated memory block at the one or more destination addresses. This store operation preferably does not perform address translation of the one or more destination addresses for reduced latency. Thus the priority mode compression does not attempt to perform memory minimization. Also, as noted above, overflow storage may be allocated in the allocated memory block, as needed.

When a requesting agent later requests the compressed data, the destination addresses are used to access the compressed data from the memory, decompress the compressed data, and provide the uncompressed data in response to the request.

1. Normal Mode Compression

In the normal compression mode (non-priority mode), the IMC 140 uses a novel memory directory for fast and efficient data retrieval during the decompression process. The novel directory procedure allows for minimum memory consumption to hold memory

allocation and directory tables, and a fixed area allocation to assist the operating system software for use in the computer main-system memory bank 110.

Memory allocation and directory maintenance is performed under control of the compression control unit 281 and the compressed data directory 271 located in the IMC 140 memory controller 220 (Figure 4). The initial address ranges and compression block sizes are set during initialization and configuration by the BIOS or boot software. The address range selection is only necessary when the system uses a plurality of requesting units with different compression formats and requirements. In a closed system where only a single client uses the memory system, a majority of this initialization can be hard wired into the standard operation. The address range and block selection flexibility gives the system more performance as required by the special needs of the requesting agents. In the PC environment for example, the PCI and AGP address ranges require separate entries in the compressed address translation table 2710. The present invention allows for multiple compressed address translation table 2710 entries for CPU to memory transactions.

In an alternate embodiment the address translation table 2710 entries can be allocated not by the operating system software but by a separate statistical gathering unit (not shown in the preferred embodiment). The statistical gathering unit monitors sequential addresses, requesting agents, and the associated block sizes and then automatically and dynamically programs entries into the compressed address translation table 2710.

In addition, if the compression operation is not required for a plurality of requesting agents or block sizes, such as graphics frame buffer or depth and texture compression, the compression address translation table 2710 is not required in the alternate embodiment.

Figure 24 - Memory Allocation

Figure 24 illustrates the preferred procedure for memory allocation within the compression and decompression system environment of the IMC 140 or alternate embodiments of the present invention. The full address bus is presented to the compressed address translation table (CATT) 2710 for address start selection, data pointer, and overflow table pointer information. The initial allocation area 2740 is a portion of system memory which has previously been allocated to a fixed size by the system or user software. The

initial allocation area 2740 receives a portion of the translated address that preferably has been translated by a simple subtraction and shift operation for look up of the first block. The initial allocation area 2740 contains one block of the compressed data for each uncompressed block in a fixed memory allocated range. Once the address for the compressed block is located, the header for the block is decoded by the compressed data header logic 2750 for determination of further decompression. The compression block header 2750 located at the front of the compressed data block determines if the block compressed to a size larger than the allocated compressed block size. If so, the overflow address translation pointer is used along with the information from the compressed header data 2750 through the select logic 2760 to select the correct overflow area pointer to read the overflow block from the overflow area 2770. The overflow area resides in the remaining portion of system memory unused by the initial allocation area. The resulting overflow block header 2790 contains information along with the original header information 2750 used by the decompression engines 550 and 555 to complete the decompression process. The output of the decompression unit is used by the output format switch 588 for selection of block information and final output as decompressed data.

Figure 26 - Memory Allocation and Initialization

Referring to the flowchart of Figure 26 and in reference to Figure 24 and the table of Figure 25, the preferred embodiment for the memory allocation and initialization is outlined. It should be noted that in Figure 24 the most recently used CATT and OAT entries could be cached by the compression controller for faster access in a system with many separately compressed memory ranges. The number of entries in the CATT is variable, and allows overflow into the memory. For faster lookup, the CATT in memory will have its entries ordered. The OAT entries are numbered so no ordering is required.

The preferred initialization 2709 is shown in figure 26. First, in step 2711 the method allocates a compressed address translation table entry. If required in step 2713, a reorder of entry data for the start and end compression block addresses is performed. In step 2715 the set method of the compression type for this memory range based on the allocate command of the initialization or operating system software. In the preferred embodiment

pages are on 4096 byte boundaries which follow the current PC architecture for address translation performed by the CPU or GART. In alternate embodiments other page sizes may be used. In addition, in other alternate embodiments the CATT may not be necessary if memory allocation is to fixed memory types such as frame buffers, or embedded appliances where a single CATT entry could describe the entire memory.

In step 2717 the method allocates a percentage of the requested memory, based on the block size and the compression type. During the allocation command sequence of step 2717 the requested compression block size and the type of compression operation performed will determine the maximum amount of allocated memory. The data (DAT) pointer is initialized in step 2719 to start at the initial block in the CATT 2710.

The overflow memory allocation and initialization in step 2721 is performed by either the initialization logic, software drivers, BIOS or operating system software. With the lossless compression algorithm used by the preferred embodiment, the maximum overflow allocation is 12.5%. Typical allocation of the overflow area in step 2770 is a portion of the original data size. For the preferred embodiment, $1/8^{th}$ the original data size is the typical choice. The overflow address table 2780 is then initialized in steps 2723, 2725 and 2727 if required. These steps initialize the headers to zero and initialize the overflow address table 2780 entries to point at the overflow address area 2770. Thus the memory allocation procedure 2709 performs the initialization of the CATT 2710 and OAT 2780, and in addition allocates the initial allocation area 2740 and the overflow area 2770.

Figure 27 – Compressed Memory Writes

Figure 27 illustrates the procedure for performing compressed memory writes. A write operation first involves a cache look-up to determine if the write data resides in the cache 291 in an uncompressed format. If so, the write data overwrites the current data in the cache 291, and this entry is marked as most recently used. In a write-back implementation, the write data is not actually written back to the system memory 110, but rather is stored only in the cache 291. In a write-through implementation, the write data is written back to the system memory 110, preferably in a compressed format, as well as being stored in the cache 291 in an uncompressed format.

If the write data does not reside in the cache 291, then an LRU block may be flushed back to the system memory, preferably in a compressed format, to free up a line in the cache 291, and the new write data is stored in the cache 291 in an uncompressed format in the freed up line. Again, this write data is not actually written back to the system memory 110 in a write-back implementation, but is written back to the system memory 110, preferably in a compressed format, in a write through implementation.

The operation of the cache 291 may also involve analysis of status bits, such as invalid and modified bits, for lines in the cache. Where the cache 291 is an L2 or L1 cache, the operation of the cache 291 may also involve analysis of status bits, such as invalid, shared, exclusive, and modified bits, for lines in the cache.

Referring to Figure 27, as write data enters the memory controller 220, a look up by the CATT 2710 is performed in step 2731 for determination of an internal cache hit. The internal compression cache 291 preferably contains normal non-compressed data. If a cache hit occurs as determined in step 2731, no compression or memory fetch of compressed block is required, and the data is retired to the cache immediately in step 2743. The uncompressed write data is preferably stored in the cache, and a most recently modified flag is set for this cache entry. In alternate embodiments the compression cache memory may be internal or external to the IMC 140 or may contain compressed data in addition to normal non-compressed data.

The write data is assembled into a decompressed block, and in the preferred embodiment, the block is stored uncompressed in the data cache. In alternate embodiments without the compression data cache, the block can be written back to the system memory 110. In the alternate embodiment, or in the case of a castout of this data from the cache, the same compressed blocks that were previously used for this uncompressed data will be reused.

If the resulting lookup of step 2731 is a cache miss, and the cache does not contain an unused line for this write data, the LRU line is selected for write back. The initial address for the write back is calculated in step 2733 using a simple subtract and shift to write the first compressed block to main memory 110. The header is read and processed, to

determine if additional blocks were previously allocated for this block of data in steps 2759 and 2735 while the write back data is compressed by the compression engine 570 or 575.

Once the compression of the data is complete, the compressed data is tested for overflow of the initial allocation block 2740 as indicated in step 2735. If larger than the initial block size, the next address allocation, step 2799 shown in Figure 29, is performed. A compressed block is stored in the block returned by the next address allocation, and the header from the next block is retrieved 2759. This loop continues until the complete compressed data is stored. If the compressed data fits without overflow it is stored in this block with an overflow indicator in the header indicating Last Block, and the test for last block of step 2741 is performed. If this block was the last one allocated previously, the store is complete. Otherwise, the header of the next block is fetched and re-written as Unused 2745. The newly fetched header is then checked for Unused, and this loop (2741, 2745) continues until all previously allocated blocks are marked unused in step 2745. The newly fetched header is then checked for Unused, and this loop steps (2741 & 2745) continues until all previously allocated blocks are marked Unused.

Figure 28 – Memory Fetch

Figure 28 illustrates the process for memory fetch 2759. As shown, in step 2751 the method determines if the data is resident in cache. If a cache hit occurs, i.e., the data resides in the cache, then data is read directly from the cache in step 2752. The cache flags are undated in step 2769 and the most recent block is marked n step 2769.

If the compressed block is not located within the cache as determined in step 2751, the initial compressed block address is calculated in step 2753. From this address the initial block is read from the system memory 110 in step 2755. In step 2757 the header instructs the memory controller 210 for the decompression process. More specifically, the method strips the header bits to determine the type of decompression, and the data is decompressed using the appropriate decompression method. In step 2761 the initial block header is tested for a last block indication to determine if the last block of the fetch has been accessed and if so marked, the process finishes with a cache invalidation of the LRU and a store of the block as MRU as in step 2769.

Thus the LRU data in the cache is removed or invalidated to make room for the newly read data, which is stored in the cache and marked as most recently used. If the header indicates additional blocks in step 2761, a fetch of the overflow block from the overflow area 2770 is required in step 2754. Based on the calculation of the overflow block pointer in step 2754 the block is read and decompressed in step 2756. In order to reduce latency, the data is sent back to the requesting agent in step 2765 and the process is ended if the last block was reached in step 2761. The book-keeping then updates the operation, setting the new cache block as MRU with a possible compression and memory write of the LRU block in cache as shown in step 2769. Thus the memory fetch operation and process of 2759 reads compressed blocks from system memory 110 decompresses these blocks and manages such cache and overflow address calculations.

Figure 29 – Next Address Generation

The next address generation shown in Figure 29 performs the calculation for the next compression block address. During step 2791 the header is examined for indications of block completion. The last/unused flag (overflow indicator) located in the header indicates completion. If the last block is not reached, the process continues with step 2702 for calculation of the next block address pointer. Once complete the next address is returned for further process. If during step 2791 the initial header indicates last block, then the process proceeds with step 2793 where the overflow process determines a new overflow address for the overflow header build. If the OAT 2780 is not full operation continues with step 2705. If the OAT 2780 entry is full a new overflow pointer is assigned in step 2795. A check for valid overflow pointer is made in step 2797 and this pointer is used if it is valid. If the overflow pointer is not valid, operation continues with the allocation of the new overflow memory block and OAT 2780 entry, step 2701. The new overflow address table 2780 pointer is set to the address of the newly allocated entry 2703. The process continues with step 2705 where the new overflow block address is calculated. Once the new block address is presented, step 2707 reads the new overflow header and based on this header step 2704 determines if the overflow block is unused. If unused is indicated in step 2704 the next sequential block's address is stored in the next address pointer in step 2706B. If a

unused in not indicated in step 2704 then the address for the next sequential block is calculated, and a return to step 2707 checks that block for unused. A reasonable implementation of the present invention for the parallel compression and decompression address allocation and data directory are shown in Figure 30. The memory allocation table, from left to right indicates the uncompressed block size, the type number entry, the initial allocation area block size, the overflow area block size, the maximum compression ratio, the initial allocation percentage of the uncompressed data, the header size without overflow, the maximum header size with overflow and sequential blocks, the maximum header size with fragmentation and non-sequential blocks, compression and fragmented data. For an average uncompressed block size of 512 bytes, the total directory size is less than 1% of the compressed data size. Thus the embedded compressed next address and overflow algorithm significantly enhances the reduction of directory information required for compression and decompression process as indicated by the present invention.

L3 Data Cache

The structured use of L3 data cache 291, which contains pre-fetched decompressed data, reduces latency by using pipelined addresses and a most recently/least recently used cache address scheme. Thus, in the preferred embodiment an L3 data cache is used to store most recently used memory pages which are read from the main memory 110. The pages are preferably decompressed by the parallel compression and decompression unit 251 and stored in the L3 cache in a decompressed format for rapid access and reduced latency. The L3 cache was discussed in detail above.

Compression Reordering

To reduce latency even further, the IMC can also operate to reorder compressed blocks for faster access of compressed data blocks. In the preferred embodiment, an optional address tag is stored in the compressed data to indicate a new byte order from the original or last byte order of the input data stream. During decompression the longest latency to recover a compressed portion of data on a compressed first block will be the last byte in the portion of the compressed block. Larger compression block sizes will increase

latency time. This method of latency reduction separates a compression block at intermediate values and reorders these intermediate values to be located at the front of the compression block. The block is reordered so that the segment most likely to be accessed in the future, e.g. most recently used, is placed in the front of the block. The tag field indicates to the decompression engine how to reorder the bytes in the intermediate segments for placement into the L3 data cache. When the block (currently stored in the L3 data cache) becomes the least recently used block, and before it is written back to main memory 110, it will be compressed with the most recently used intermediate segment at the front of the compressed block before storage back into the main memory 110. This method of latency reduction is especially effective for program code loops and branch entry points and the restore of context between application subroutines. In an alternate embodiment, a tag field could be present for each intermediate block such that the new compression order of intermediate segments track the N most recent intermediate blocks in the order in which they were accessed over time. In the preferred embodiment only the block header will indicate which intermediate block segment is first in the recompression and restore process, the order will then follow the nature of the original data stream.

Figure 31 illustrates how out of order compression is used to reduce read latency on subsequent reads from the same compressed block address. The original compressed block 2510 is stored in main memory 110 in the order written by the requesting agent. As the requesting agent issues a new request, the steps that are indicated in sequence 2530 are preformed. At the time compressed block 2510 is ready to be re-compressed for storage into the main memory 110, an out of order flag is attached to the header field indicating that the intermediate blocks are out of order from the original written order. The new compressed out of order block 2520 is written back to main memory 110.

Variable Compression Block Size

In the preferred embodiment, the compression block size, representing the input data block before compression, is dynamic and can be adjusted in size to reduce latency of operation. For example, the local bus interface 106 may compress with input blocks of 32 or 64 bytes while video 235 or graphics engine 212 may compress with input blocks of 256

or 512 bytes. In the preferred embodiment the power-on software will set default block sizes and compression data formats for each of the requesting units and for specific address ranges. Also, the preferred embodiment includes software control registers (not shown) that allow interface software drivers to dynamically adjust the compression block sizes for a plurality of system memory performance levels. Thus, by dynamically adjusting the compression block sizes based on one or more of the requesting agent, address range, or data type and format, latency can be minimized and overall efficiency improved.

Dynamically Gather Statistics to Adjust Block Size

In one embodiment, the IMC 140 may gather statistics to dynamically adjust block size. The IMC gathers statistics on sequentiality of addresses and locality of addresses. In this embodiment, the IMC 140 includes a statistical unit which analyzes, for example, address blocks, localities of requests to the same page or block, and the sequentiality of the addresses being accessed.

Lossless Decompression

One embodiment of the parallel decompression engine 550 for the lossless decompression of compressed data is now disclosed. Data compression methods may include serial compression methods, where only one symbol from the uncompressed data is examined and compressed at a time, and the novel parallel compression methods described above, where a plurality of symbols from the uncompressed data may be examined and compressed at a time. In one embodiment, the parallel decompression engine 550 may be able to decompress data compressed by serial or parallel decompression methods. Likewise, decompression of compressed data using the parallel decompression technologies of the present invention produces the same uncompressed data stream as decompression of the same compressed data using prior art serial decompression techniques. The compressed data created using the parallel compression methods described above is designed to be identical to compressed data created using serial compression algorithms; therefore, decompressing data compressed with the parallel method described above by either serial or parallel decompression engines will result in the same uncompressed data. Preferably,

decompression is performed as fast as the compression operation or faster. Also, in alternate embodiments, decompression engines 550/555 may be placed in a plurality of locations within the system or circuit. Multiple decompression engines allow for a custom operation of the decompression process and a custom bandwidth or throughput may be designed depending on the number of stages used in the decompression engine 550. Therefore, below is a parallel decompression algorithm for the parallel decompression engine 550 that yields higher bandwidth than prior art serial algorithms.

Figures 32 - 43 – An Embodiment of a Parallel Decompression Engine

The parallel decompression engine 550 may be divided into a series of stages, preferably pipelined stages. The stages of the decompression engine 550 are illustrated in Figure 33. As shown, the decompression engine 550 may include a first stage 25501 comprising decoders, a second stage 25505 comprising preliminary (also called initial or primary) select generation logic, a third stage 25509 comprising final select generation logic, and a fourth stage 25513 comprising data selection and output logic. A pipe register 25503 may be coupled to the first stage 25501 and the second stage 25505. A pipe register 25507 may be coupled to the second stage 25505 and the third stage 25509. A pipe register 25511 may be coupled to the third stage 25509 and the fourth stage 25513. According to one embodiment, the pipelined design is expected to utilize four stages to run at 133 MHz using a 0.25μ CMOS technology. These stages are preferably divided up, or alternatively combined, as the silicon process technology requires. Only the last stage in this pipeline 25513 uses the history window, and that final stage contains minimum logic. Based on this, this function may be extended to more than four stages if a significantly faster clock is available. Thus, in alternate embodiments, as processing improves and clock rates increase, the stages of the decompression engine 550 may increase to raise the decompression rate with the same input compression stream. However, for the preferred embodiment the four stages shown are the logical divisions of the function. Other embodiments may include fewer than four stages. For example, a three-stage embodiment may combine the second and third stage into one stage.

In the preferred embodiment, the decompression engine 550 includes a pipelined, multi-stage design. The pipelined, multi-stage design of the decompression engine 550 enables the substantially simultaneous or concurrent processing of data in each stage. As used herein, the term "decompression cycle" includes operation of all stages of the pipeline on a set of data, from analysis of tokens in an input section of data in the first stage to production of output uncompressed data in the last stage. Thus, multiple "decompression cycles" may be executing substantially concurrently, i.e., different stages of multiple "decompression cycles" may be executing substantially concurrently.

For example, the first stage 25501 may receive a first plurality of codes (also called tokens), and load the first tokens into the decoders at the start of a first decompression cycle. The decoders may extract various first decoded information from the first tokens, and this first decoded information may be latched into pipe register 25503. The first decoded information may then be loaded into the preliminary select logic of the second stage 25505. While the preliminary select logic of the second stage 25505 is operating on the first decoded information, a next plurality of tokens (second tokens) may be received by the first stage 25501 and loaded into and processed by the decoders at the start of a second decompression cycle, substantially simultaneously, to produce second decoded information. When stage two has completed generating preliminary selects from the first decoded information in the first decompression cycle, the preliminary selects are latched into pipe register 25507 in the second decompression cycle. Similarly, when stage one has completed generating the second decoded information in the second decompression cycle, this second decoded information may be latched into pipe register 25503. The preliminary selects may then be loaded into the third stage 25509 for resolution into final selects, while the second decoded information generated in the first stage 25501 for the second decompression cycle is loaded into the second stage 25505, and a next (third) plurality of tokens is received by the first stage 25501 and loaded into the decoders to begin a third decompression cycle. Thus, in the four-stage embodiment of decompression engine 550, four decompression cycles may be active in the decompression engine 550 substantially simultaneously.

As used herein, in the context of the first stage examining a plurality of tokens from the compressed data in parallel in a current decompression cycle, the term "in parallel"

includes the notion that a plurality of tokens may be operated on by the logic during a single pipeline stage of the decompression engine 550. The term "in parallel" also may include the notion that a plurality of decoders operate on a plurality of tokens during a single pipeline stage of the decompression engine 550. The plurality of tokens may actually be extracted from the input data section serially or consecutively. The plurality of tokens may then be assigned to available decoders as they are extracted from the input data section. Once tokens have been assigned to available decoders, portions of the processing of the tokens by the decoders may be performed in parallel. In addition, the term "in parallel" may also include the notion that a plurality of decoders output decoded information in parallel to the next stage of the pipeline.

As used herein, in the context of generating a plurality of selects in parallel, the term "in parallel" includes the notion that the select logic (stages 2 and/or 3) may concurrently process decoded information corresponding to a plurality of tokens substantially concurrently and/or the select logic may operate to generate selects for a plurality of output uncompressed symbols substantially concurrently. As described below, the select logic shares information regarding the selects that are being generated in parallel for different output uncompressed symbols.

Therefore, in general, information for decompressing more than one token may be loaded into a stage, operations of the stage performed on the tokens, and the results for all the tokens may then be latched out of the stage into a pipe register for processing in the next stage. In each stage, there may be copies of the logic for performing substantially simultaneous operations "in parallel" on a plurality of inputs.

For example, in the first stage 25501, an extracted token is assigned to one decoder. In the second, third, and fourth stages, there may be one copy of the logic for performing the operations of the stage for each potential output byte. Note that some operations in some stages may have dependencies that may utilize sequential processing. For example, loading a second token in a second decoder of the first stage 25501 may utilize count and other information generated by the loading of a first token in a first decoder.

To understand this novel decompression, the table of Figure 32 illustrates the compression mask and index-coding algorithm for a sample code. In alternate embodiments, other codes may alter the design of the decompression unit. One embodiment may include all the codes included in Figure 32 except the code for compressing one byte that uses 8 bits. In compressed input data, a code may also be referred to as a "token."

With the codes shown in the table of Figure 32, the decompression tree in Figure 34 allows decoding of at most 8 bytes of the input in one cycle. In this example, at most 8 bytes (64 bits) are extracted from the compressed data as input data to the decompression engine of Figure 33 for each decompression cycle. The smallest encoded data is 8 bits, so the minimum number of decoders (25521-25535), indicated in Figure 34, for 8 bytes is 8 (64 bits/8 bits). Each of these decoders could see one of many data inputs depending on the prior compressed data.

Figure 34 illustrates the decoder stage 25501, which is the first stage of the decompression engine of Figure 33. The decompression tree, shown in Figure 34, utilizes very fast decoding at each stage to determine the proper data for the next stage. The Window Index, Start Count and Data Byte output (Figure 32) are preferably latched for the next stage of the decode pipeline of Figure 33. This decode pipeline requires the assembly of the output data. More detail of the preferred Decode block can be seen in Figure 35.

Figure 35 illustrates the logic of one of the first stage decoders of Figure 34. In Figure 35, the Check Valid block 25553 verifies that enough bits are available for the checker 25555(a-e). After extracting one or more codes from the input data to be decoded by one or more decoders, there may not be enough bits in the input data to construct a complete token. For example, in the decompression engine described above that accepts 8 bytes (64 bits) of input data in a cycle, if six 10-bit codes are loaded into the first six decoders, four bits would be left in the input data, not enough to construct a complete token. In another example using 64-bit input data, if four 10-bit codes and one 13-bit code are loaded into the first five decoders, 11 bits are left in the input data. The Check Valid block 25553 may then check the flag information in the 11 bits to determine if there is a complete code in the 11 bits (an 8, 9 or 10 bit code). If there is a complete code, then the code is

loaded in the next decoder. If the flag information indicates that the 11 bits are an incomplete code longer than 11 bits (a 13 or 25 bit code), then the bits are not loaded and examined until a later decode cycle. The tables for the Check Valid blocks are illustrated in the tables of Figures 36a and 36b. In the preferred embodiment, the longest path through Check Valid 25553 should be 3 gates, and the Byte Check 25555(a-e) will only add one gate because the check is an output enable. The outputs from the Check Valid logic 25553, and the Byte Check logic 25555 in Figure 35 show 0 as the most significant bit, and 6 as the least significant bit.

The data generate logic 25557 is a multiplex of the input data based on the check select 25555 input. At most, one Byte Check 25555 should be active for valid data. An alternate embodiment may include a checker that is added to this decoder to verify that one byte check is active for valid data. The table of Figure 36b describes the Data Generate outputs based on the Data Input and the Byte Check Select for codes similar to those illustrated in Figure 32.

Referring again to Figure 33, the second stage 25505 of the decompression engine 550 begins calculating pointers (also called "selects") to the appropriate bytes from the history window for compressed data that have been latched in the 168-bit pipe register 25503. For each decoder, stage two receives an index, a start count, an index valid bit, a data byte and a data byte valid bit. In the embodiment of Figure 33, stage two would receive eight indexes, eight start counts, eight index valid bits, eight data bytes, and eight data byte valid bits, one from each of the eight decoders in stage one. In one embodiment, the data byte is passed through without being used by stage two. In one embodiment, the indexes, start counts, index valid bits, and data byte valid bits from each of the decoders are duplicated to the preliminary select logic for each of the output bytes of stage two. Thus, in the embodiment of Figure 33, the preliminary select logic for each of the 16 output bytes receives the index, start count, index valid bit, and data byte valid bit from each of the eight decoders in stage one.

With minimal logic, a preliminary select may be calculated for each of the 16 output bytes of stage four 25513. The preliminary selects are latched in the 144-bit pipe register 25507. Each select latched into 25507 is a 7 bit encode (for a 64-entry window) with a

single bit overflow. These signals are latched 25507 and used by the next unit 25509 in stage three. In one embodiment, the selects will have the values of 0-63 if a window value is to be used for this output byte, 64-71 if one of the eight data bytes is to be used for this output byte, and an overflow if the data for this output byte is a result of one or more of the other parallel decodes occurring with this data. The third stage 25509 checks each of the overflows from the previous stage 25505. If inactive, the 7 bit select is passed on unchanged. If active, the select from the correct stage two decoder 25505 is replicated on the select lines for this output byte.

The final stage of the decompression, stage four 25513 as illustrated in Figure 33, selects the data from the history window or the data bytes passed from the first stage to build the output data. The output bytes that are assembled are then added to the history window for the next decode cycle.

In one embodiment, the first stage may consider the number of output bytes when decoding codes from the input data in a cycle. For example, the maximum output of the embodiment of Figure 33 is 16 bytes per cycle. If a first code being decoded in a first decoder represents more than 16 output bytes, then the first stage 25501 may leave the first code loaded in the first decoder for as many cycles as it take to decompress all of the output bytes represented by the first code. Other codes that may be loaded in the other decoders are not decoded until there are available output data bytes to serve as destinations for the uncompressed symbols to be generated from the tokens. For example, if the first code loaded in the first decoder represents 24 output bytes, then 16 of the 24 output bytes may be decoded in a first cycle, and the remaining 8 in a second cycle. This leaves 8 output bytes for the other codes in the other decoders. Additionally, the last stage 25513 may include data valid bits so that the proper output data assembly can occur if fewer than 16 bytes can be decoded for any one cycle.

Figure 37 - Calculating initial selects and overflows

Figure 37 illustrates logic 26012 for calculating initial selects and overflows according to one embodiment of the invention. In one embodiment, this logic is included in the second stage 25505 of the decompression engine as illustrated in Figure 33. In one

embodiment, there is one logic 26012 in the second stage for each output byte in the decompression engine 550. For example, in the decompression engine illustrated in Figure 33, there would be 16 of the logic 26012 in stage two, one for each output byte. Logic 26012 begins the calculation of pointers to the appropriate bytes from the history window for compressed data which have been latched in the 168-bit pipe register 25503 from the first stage in Figure 33. In the embodiment shown in Figure 37, each logic 26012 in stage two receives a copy of the Index 26006 and Count 26000 from each decoder in stage one. Each logic 26012 in stage two also receives a Data Byte Valid bit 26002 and an Index Valid bit 26004 from each decoder.

With minimal logic, a preliminary select 26010 may be calculated in stage two for each of the output bytes, and the preliminary selects 26010 may then be latched in the 144-bit pipe register 25507 of Figure 33. For example, each preliminary select may be a 7 bit encode (for a 64-entry window, plus eight data bytes) with a single bit overflow 26008. Embodiments with other sizes of history windows and/or other numbers of data bytes may require a different number of bits and a different numbering scheme for the preliminary selects. The preliminary selects 26010 are latched into 25507 and used by the next unit 25509 in stage three as shown in Figure 33. The selects may have the values of 0-63 if a window value is to be used for this output byte or the values of 64-71 if one of the eight data bytes is to be used for this output byte. The overflow bit 26008 may be set if the data for the preliminary select 26010 is a result of one or more of the other parallel decodes occurring with this data. In this case, the index may be used in stage three to resolve the preliminary select by copying the appropriate select from another output byte to the select for this output byte.

Other embodiments may use history windows of various sizes, for example, from 32 entries to 4096 (or greater) entries. The size of the history window may be determined by the number of gates available for the design, the timing of stage four, and the compression ratio desired. More history window entries may typically yield a better compression ratio. As the history window size changes, the size of the index, preliminary and final selects may also change. For example, a history window with 2048 entries would require an 11-bit index, 13-bit preliminary select (11 bits for the index, one bit to

indicate data byte, one bit to indicate overflow), and 12-bit final select (11 bits for the index, one bit to indicate data byte).

In one example of a decode where an overflow bit may be set, a first decoder may decode a first token and output a pointer to a first data byte, and a second decoder may decode a second token and output a pointer to a second data byte. A third decoder may decode a third token that represents a compressed string including the first and second data bytes from the first and second tokens. As these data bytes are not in the history window yet, the overflow bit 26008 is set to signify that the data for the third decoder's output byte is defined by one of the prior decoders in the current decode. The preliminary select output of the second stage for the third decoder is resolved into a final select in the third stage. In this example, two preliminary selects may be generated for the third token; the first pointing to the first decoder's output byte, and the second pointing to the second decoder's output byte.

In Figure 37, if the preliminary select is for a data byte, the overflow bit 26008 will not be set, the most significant bit (bit 6) will be set, and bits 0-2 may be used to specify which of the eight data bytes the output byte refers to. If the preliminary select is for a window byte, the overflow bit 26008 will not be set, the most significant bit (bit 6) will not be set, and bits 0-5 may be used to specify which of the 64 window bytes the output byte refers to. If the overflow bit is set for the preliminary select, then bits 0-6 may specify which of the preliminary selects prior to this preliminary select is to be used to locate the data for this preliminary select.

In Figure 37, N is the output byte number for the logic 26012. In this example, there are 16 output bytes, so N is an integer between 0 and 15. In this example, there are eight decoders in the first stage. One start count 26000, one index 26006, and one data byte valid bit and one index valid bit are input from each decoder. The start count for a decoder is calculated in stage one by adding the number of output bytes to be generated on this decoder to the input number of output bytes to be generated on all previous decoders (i.e. the start count for the previous decoder). For example, suppose there are four decoders (0-3), and decoder 0 is loaded with a code to decode 1 output byte, decoder 1 is loaded with a code to decode 3 output bytes, decoder 2 is loaded with a code to

decode 4 output bytes, and decoder 3 is loaded with a code to decode 2 output bytes. The start count for decoder 0 is (0+1) = 1. The start count for decoder 1 is (1+3) = 4. The start count for decoder 2 is (4+4) = 8. The start count for decoder 3 is (8+2) = 10.

Block 26001 of Figure 37 compares the input start counts for the decoders with N (the output byte number for this logic 26012). Block 26001 chooses the last decoder with a start count <= N. For example, if the eight start counts 26000 in Figure 37 from decoders 0-7 are (1,3,6,7,11,14,15,20), and N=9 (this is the 10^{th} output byte), then decoder 4 (start count = 11) would be chosen. This serves to choose the decoder from which this output byte is to be generated.

In this example, block 26001 outputs a 3-bit encoded decoder number and an 8-bit decoded version of the decoder number. The 8-bit decoded version is output to selects 26003, 26005, and 26007, where it is used to select the data byte valid bit 26002, index valid bit 26004, and index 26006 for the decoder generating this output byte.

If the data byte valid bit 26002 for the selected decoder is set and the index valid bit 26004 for the selected decoder is clear, then the encoded 3-bit decoder number is output on bits 0-2 of the preliminary select 26010 (the least significant bits), and bit 6 (the most significant bit) is set to indicate that the preliminary select is for a data byte. Note that for the 64-entry history window and eight data byte embodiment previously described, the data byte select value is in the range 64-71 to select one of the eight data bytes.

If the index valid bit 26004 for the selected decoder is set and the data byte valid bit 26002 for the decoder is clear, then bit 6 (the MSB) of the preliminary select 26010 is cleared. The output byte number N is subtracted from the index 26006 from the selected decoder, and the resulting adjusted index is output on bits 0-5 of preliminary select 26010. By way of example, consider a decompression engine with eight input bytes, eight decoders (0-7), sixteen output bytes (0-15), and a 64-entry history window (0-63). If decoder 0 is decoding a code generating four output bytes, then logic 26012 for output byte 0 will generate the preliminary select for the first byte of the four output bytes being generated from the code on decoder 0. If the index 26006 from decoder 0 is 16, then 16 – 0 = 16. This means that the first byte of output from the code being decoded on decoder

0 is to come from entry 16 in the history window, where entry 0 is the most recent entry and entry 63 is the oldest entry. Logic 26012 for output byte 1 will generate the preliminary select for the second byte of the four output bytes being generated from the code on decoder 0. The second byte's preliminary select is 16 – 1 = 15. The second byte of output from the code being decoded on decoder 0 is to come from entry 15 in the history window. Continuing, the preliminary selects for the third and fourth output bytes, being generated on logic 26012 for output bytes 2 and 3, are 14 and 13, respectively.

It is possible for a preliminary select being generated in a logic 26012 to be for data being generated in the current decompression cycle, and thus the data for the output byte will not yet be in the history window. In this case, subtracting the output byte number N from the index will produce a negative result, and overflow bit 26008 will be set for the preliminary select. For example, if decoder 3 is decoding a code generating three output bytes, output byte 5 is the next available output byte, and the index for decoder 3 is 1, then logic 26012 for output byte 5 will generate a preliminary select of 1 – 0 = 1, logic 26012 for output byte 6 will generate a preliminary select of 1 – 1 = 0, and logic 26012 for output byte 7 will generate a preliminary select of 1 – 2 = -1. The –1 preliminary select indicates that the data for the output byte is to come from the first output byte of the current decompression cycle. The overflow bit for output byte 7 will be set to indicate that this preliminary select is for data that is not yet in the history window. The preliminary select outputs on bits 0-5 will indicate which of the preliminary selects in the current decompression cycle points to the data for this preliminary select.

In one embodiment of logic 26012, data byte valid bit 26002 and index valid bit 26004 are NOR'd, and the output of the NOR is OR'd to bits 5 and 6 of the preliminary select. If both valid bits are 0 for a decoder, then bits 5 and 6 will be set for the preliminary select. Note that in the embodiment with 64 history window entries and eight data bytes, values above 71 are not valid selects. Thus, in this embodiment, a preliminary select for an output byte with bits 5 and 6 set may be used to indicate that no data is being generated for the output byte in this decompression cycle. Other embodiments with different history window sizes, number of data bytes, and/or number of output bytes may

use other invalid select values to indicate that no data is being generated for an output byte in a decompression cycle.

Figure 38 - Converting preliminary selects into final selects

Figure 38 depicts one embodiment of a third stage of a decompression engine 550 such as stage three 25509 of Figure 33. The third stage checks the preliminary selects 26050 for the output bytes from the previous stage. If the overflow bit (26008 of Figure 37) of a preliminary select is not set, the 7-bit select for the output byte (bits 0-6 of preliminary select 26010 of Figure 37) is passed to the next stage unchanged. If the overflow bit is set, this indicates that the data for this output byte is being generated in the current decompression cycle. The data for the preliminary select will be pointed to by one of the previous output byte selects in the current decompression cycle. The select for the previous output byte is replicated on the select lines for this output byte. Note that the overflow bit for the first select (preliminary select 0) will not be set, because there are no "prior" selects in the current decode for the select to refer to. Thus, preliminary select 0 passes through stage three unchanged as final select 0. Final select 0 is input into the logic for resolving each of the preliminary selects subsequent to select 0 (preliminary selects 1 through N-1). Final select 0 and preliminary select 1 are input into the logic for resolving preliminary select 1. If the overflow bit for preliminary select 1 is not set, then preliminary select 1 is passed through unchanged as final select 1. If the overflow bit is set, then the final select 0 is passed through as the final select for select 1. Final selects 0 and 1 and preliminary select 2 are input into the logic for resolving preliminary select 2. If the overflow bit for preliminary select 2 is not set, then preliminary select 2 is passed through as final select 2. If the overflow bit is set, then preliminary select 2 is used to determine which of the input final selects (0 and 1) is to be output as final select 2. In general, this procedure is followed for all of the N input preliminary selects. Thus, the input to the logic for resolving preliminary select N-1 includes the final selects for selects 0 through N-2, and preliminary select N-1. If the overflow bit is not set for preliminary select N-1, then preliminary select N-1 is passed through unchanged as final select N-1. If the overflow bit is set, then the

contents of preliminary select N-1 are used to determine which of the input final selects is to be used as the value for final select N-1.

Figure 39 - Generating uncompressed output bytes from generated selects

Figure 39 depicts one embodiment of a fourth stage of a decompression engine 550 such as stage four 25513 of Figure 33. In stage four, the final selects 26068 output from the third stage as depicted in Figure 38 are used to assemble the output bytes 26070 by selecting bytes from the history window 26062 or the data bytes 26064 passed from the first stage. In this embodiment, each output byte selector 26066 may select from one of 64 bytes (0-63) in history window 26062 or from one of eight bytes (64-71) in data bytes 26064. In one embodiment history window 26062 and data bytes 26064 may be combined in a combined history window 26060. In other embodiments, the data bytes and history window may be maintained separately. The final selects 26068 are indexes into either the history window 26062 or the data bytes 26064 passed from stage one. The output bytes 26070 that are assembled may be sent to the output data stream (appended to the end of the output bytes from any previous decompression cycles) and may be inserted in the history window for the next decode cycle. Stage four may also include a data valid bit (not shown) for each of the output bytes 26070 so that the proper output data assembly may occur if fewer than the maximum number of bytes (16 in this embodiment) are to be decoded in a decode cycle. In one embodiment, an invalid index value in a final select for an output byte may clear the data bit to indicate that the output byte does not contain valid data in this decompression cycle. Output bytes that are not valid may not be sent to the output data or written in the history window.

Figure 40 – Data flow through a decompression engine

Figure 40 illustrates data flow through one embodiment of a decompression engine 550. The decompression engine 550 receives a compressed input stream 1000. The compressed input stream 1000 is then decompressed in one or more decode (or decompression) cycles, resulting in a decompressed output stream.

As a first step 1002 of a decompression cycle, from 1 to N tokens from the compressed data stream 1000 may be selected for the decompression cycle and loaded in the decompression engine 550, where N is the maximum number of decoders in stage one. The tokens are selected serially from the first token in the data stream 1000. In one embodiment, a section may be extracted from the compressed data stream 1000 to serve as input data for a decompression cycle, and the tokens may be extracted from the extracted section. For example, in one embodiment, a section of four bytes (32 bits) may be taken, and in another embodiment, a section of eight bytes (64 bits) may be taken. In one embodiment, steps 910 through 920 as illustrated in Figure 43d may be followed to select the 1 to N tokens for the decompression cycle. In one embodiment, a token may be selected from the input data stream 1000 for the decompression cycle if 1) there is a decoder available (i.e., one or more decoders haven't been assigned a token to decode in the decompression cycle); and 2) the remaining bits in an input section of the compressed data comprise a complete token (after extracting one or more tokens from the input data, the remaining bits in the input data may not comprise a complete token). If any of the above conditions fails, then the decompression cycle continues, and the last token being examined (the one that failed one of the conditions) is the first token to be loaded in the next decompression cycle. Preferably, no correctly formatted token is ever totally rejected; i.e., any token presented to the decompression cycle as a first token considered for the decompression cycle will meet all the conditional requirements. In other words, 1) a decoder will always be available at the start of a decompression cycle; and 2) the input data size in bits is at least as big as the largest token size in bits.

Once the 1 to N tokens for the decompression cycle are selected in the first step 1002, the 1 to N tokens are passed into stage one 1006 for decoding. In one embodiment, step 1002 may be performed as part of stage one of the decompression engine 550. In one embodiment, one token is assigned to one decoder, and one decoder may process one token in a decompression cycle. Stage one may include N decoders. There are preferably at least enough decoders to accept a maximum number of tokens that may be in the input data. For example, if the input data is 32 bits, and the minimum token size is 9 bits, then there are preferably at least three decoders. Preferably, the number of decoders equals the maximum

number of tokens in the input data. Figure 34 illustrates an embodiment of decompression engine 550 with eight decoders. Figures 41-42 illustrate an embodiment of decompression engine 550 with three decoders. Figure 35 illustrates an embodiment of a decoder. The decoders of stage one 1006 decode the input tokens into start counts, indexes, index valid flags, and data valid flags, with one copy of each from each decoder being passed to stage two 1008 for each of the X output bytes to be generated in the decompression cycle. The 1 to N original input data bytes are passed from stage one to the combined history window 1014. A data byte is valid only if the token being decoded on the decoder represents a byte that was stored in the token in uncompressed format by the compression engine that created the compressed data. In this case, the uncompressed byte is passed in the data byte for the decoder, the data byte valid bit for the decoder is set, and the index valid bit for the decoder is cleared.

Stage two 1008 takes the inputs from stage one 1006 and generates preliminary selects for 1 to X output bytes, where X is a maximum number of output bytes that may be decoded in one decompression cycle. Stage two 1008 also generates an overflow bit for each preliminary select. Stage two then passes the preliminary selects and overflow bits to stage three 1010. Stage three 1010 inspects the overflow bit for each of the preliminary selects. If the overflow bit of a preliminary select is not set, then the contents of the preliminary select point to one of the entries in the history window 1014 if the index valid bit is set for the output byte, or to one of the data bytes passed from stage one 1006 to the combined history window if the data byte valid bit is set for the output byte. Preliminary selects whose overflow bits are not set are passed to stage four 1012 as final selects without modification. If the overflow bit is set, then the contents of the preliminary select are examined to determine which of the other preliminary selects is generating data this preliminary select refers to. The contents of the correct preliminary select are then replicated on this preliminary select, and the modified preliminary select is passed to stage four 1012 as a final select. In one embodiment, a preliminary select with overflow bit set may only refer to prior preliminary selects in this decompression cycle. For example, if the overflow bit for the preliminary select for output byte 3 is set, then the preliminary select may refer to data being generated by one of preliminary selects 0 through 2, and not to

preliminary selects 4 through (N-1). In one embodiment, stages two and three may be combined into one stage.

Stage four 1012 uses the final selects it receives from stage three 1010 to extract byte entries from the combined history window 1014. The final selects may point to either history window bytes or data bytes passed from stage one 1006. The number of bits in a final select are determined by the number of entries in the history window plus the number of data bytes. For example, a 64-byte history window plus eight data bytes totals 72 possible entries in the combined history window, requiring seven bits per final select. Other history window sizes and/or number of data bytes may require different final select sizes. Stage four 1012 extracts the data from the combined history window and constructs an output of between 1 and X uncompressed output data bytes 1016. Stage four 1012 may use a data valid flag for each of the X output data bytes to signal if a data byte is being output for this output data byte in this decompression cycle. The data valid flags are necessary because it may not always be possible to decompress the maximum amount of output bytes (X) in a decompression cycle. The output bytes 1016 may then be appended to the output data stream and written into the history window 1014. In one embodiment, if the history window is full, the oldest entries may be shifted out of the history window to make room for the new output bytes 1016, or alternatively the history window may be stored in a ring buffer, and the new entries may overwrite the oldest entries. The decompression cycle may be repeated until all of the tokens in the input stream 1000 are decompressed.

Figure 41 - Three decoder stages to accept 32 bits of input data

Figure 41 illustrates an embodiment of a stage one with three decoders. The embodiment is similar to the embodiment with eight decoders shown in Figure 34. For the embodiment shown in Figure 41, the input data 1100 will comprise four bytes (32 bits). The compressed data will be encoded with codes similar to those shown in Figure 32, but the 8-bit code for compressing one byte is not allowed. Thus, the minimum token, or code, size is 9 bits, for a token representing one uncompressed byte. The input data 1100 of Figure 41 may include at most three complete tokens (32/9 = 3, with 5 bits remaining).

Thus, this embodiment requires three decoders to accept the maximum number of tokens that can be extracted from the input data for a decompression cycle.

In this embodiment, bits D0:D24 are passed to decoder 0 1102. Decoder 0 1102 examines the flag field of the token starting at D0 to determine the bit size of the token. Decoder 0 1102 then passes the bit size to 1104, which passes bits E0:E22 (23 bits, the number of bits in the input data 1100, 32, minus the smallest token size, 9) to decoder 1 1106. The 23 bits may include bits D9:D31 if decoder 0 1102 is decoding a 9-bit token, bits D10:D31 if decoder 0 1102 is decoding a 10-bit token, or bits D13:D31 if decoder 0 1102 is decoding a 13-bit token. If decoder 0 1102 is decoding a 25-bit token, then the remaining seven bits do not contain a complete token, so no bits are passed to decoder 1 1106 from 1104 in this decode cycle, and the number of bits passed to decoder 1 1106 from decoder 0 1102 (25) indicates to decoder 1 1106 that it is not to be used in this decode cycle. If decoder 1 1106 receives bits from 1104, decoder 1 1106 examines the flag field of the first token in the bits. If the flag field of the token indicates that the token is a 25-bit token, then the token is not complete, and decoder 1 1106 and decoder 2 1110 are not used in this decompression cycle. If the flag field of the token indicates that this is a 9, 10 or 13-bit token, then the token is loaded in decoder 1 1106, and the total number of bits used is passed to 1108 and to decoder 2 1110. 1108 passes bits F0:F13 (14 bits, the number of bits in the input data 1100, 32, minus two times the smallest token size, 9) to decoder 2 1110). The 14 bits may include bits E9:E22 if decoder 1 1106 is decoding a 9-bit token, bits E10:E22 if decoder 1 1106 is decoding a 10-bit token, or bits E13:E22 if decoder 1 1106 is decoding a 13-bit token. Decoder 2 1110 may then examine the flag field of the token starting at F0 to determine the token size. Decoder 2 1110 may then compare the token bit size with the remaining number of bits (determined from the input bits used by the first two decoders) to determine if the token is complete. If the token is complete, then the token is loaded in decoder 2 1110 for decoding in this decompression cycle. If the token is not complete, then decoder 2 1110 is not used in this decompression cycle.

A few examples of loading tokens are given to illustrate the loading process. If input data 1100 includes a 25-bit token starting at bit 0 (D0), then only seven bits are left in input data 1100 after decoder 0 is loaded with the 25-bit token. In this case, decoders 1 and

2 are not loaded with tokens in this decompression cycle. If decoder 0 is loaded with a 9, 10 or 13-bit token, and the remaining bits in input data 1100 are an incomplete 25-bit token (as determined from the flag field in the incomplete token), then decoders 1 and 2 are not loaded in this decompression cycle. Other combinations of tokens in input data 1100 may result in decoders 1 and 2 being loaded or in all three decoders being loaded for a decompression cycle.

Figure 42a - A decompression engine with four input bytes, three decoders, and four output bytes

Figure 42a illustrates an embodiment of decompression engine 550 with four input bytes 1120 comprising 32 bits, three decoders in stage one 1122, and four output bytes 1136. This embodiment is suitable for decoding codes (tokens) similar to those depicted in Figure 32, excluding the 8-bit code used to encode one compressed byte. Figure 42a illustrates that in stage two 1126, stage three 1130, and stage four 1134, there is parallel logic for generating each of the output bytes (in this embodiment, four output bytes).

One or more tokens are extracted from input bytes 1120 and loaded into decoders in stage one 1122. The tokens are decoded by the decoders, and start count, index, index valid and data valid information 1124 is passed to stage two 1126. Data byte information (not shown) may also be produced for the decoders and passed through for use in stage four 1134. The information 1124 from each decoder is copied to the stage two logic for each output byte. Stage two 1126 generates preliminary selects 1128 from the information 1124 passed in from stage one 1122. Stage two 1126 passes the preliminary selects to stage three 1130. Stage three 1130 generates final selects 1132 from the preliminary selects 1128 passed from stage two 1126. As shown, the final select 1132 generated on a stage three logic 1130 for an output byte is passed to the stage three logic for all subsequent output bytes. This allows a preliminary select 1128 with overflow bit set indicating that the data for the output byte is being generated in the current decompression cycle to be resolved by copying the final select for the correct output byte to be used as the final select for this output byte. The final selects 1132 are passed to stage four 1134. Stage four 1134 uses index information in the final selects 1132 to select entries from the history window (not

shown) or the data bytes passed from the decoders in stage one 1122 and copies the selected data into output bytes 1136. The output bytes 1136 may then be written to the output data (not shown), and may also be written into the history window as the latest history window entries.

Used Data Calculation logic 1123 in stage one may be used to maintain a count of output bytes being generated in the current decompression, and also to maintain a count of the number of tokens being decoded and decompressed in the current decompression cycle. This information is used in stage one for shifting the compressed data prior to extracting the input bytes 1120 in a later decompression cycle. Used Data Calculation logic 1123 is further explained by the example decompression cycles described in Figure 42b.

Figure 42b – An example decompression

Figure 42b is used to illustrate an example decompression of an input to an embodiment of decompression engine 550 as illustrated in Figure 42a. In this example, three tokens have been extracted from input bytes 1120. The first token, a10-bit token representing two compressed bytes, is loaded in decoder 0. The second token, a 10-bit token representing three compressed bytes, is loaded in decoder 1. The third token, a 9-bit token representing one uncompressed byte, is loaded in decoder 2. Decoder 0 generates the information (start count = 2, index = i0, index valid = 1 (true), data valid = 0 (false)) for the first token. The start count (2) is passed to decoder 1. Decoder 1 generates the information (start count = 5, index = i1, index valid = 1, data valid = 0) for the second token. The start count is the sum of the output byte counts for decoder 0 and decoder 1 (2 + 3 = 5). The start count (5) is passed to decoder 2. Decoder 2 generates the information (start count = 6, index = d2, index valid = 0, data valid = 1) for the third token. In this example indexes starting with (i) are to entries in the history window, and indexes starting with (d) are in the data bytes.

Stage two 1126 uses the information 1124 generated from the decoders in stage one 1122 to generate preliminary selects for the four output bytes. Two output bytes are being generated from the first token in decoder 0. The stage two logic for output byte 0 examines the information 1124 and determines that it is to generate a preliminary select 1126 for the

first byte compressed in the first token. The preliminary select output 1128 for output byte 0 is index = i0. The stage two logic for output byte 1 examines the information 1124 and determines that it is to generate a preliminary select 1126 for the second byte compressed in the first token. The preliminary select output 1128 for output byte 0 is index = (i0 – 1). The output byte number is subtracted from the original index to generate the actual index number for this output byte. Thus, preliminary selects for all output bytes to be produced from the first token are generated for the first two output bytes. The stage two logic for output byte 2 examines the information 1124 and determines that it is to generate a preliminary select 1126 for the first byte compressed in the second token. The preliminary select output 1128 for output byte 2 is index = (i1 – 2). The stage two logic for output byte 3 examines the information 1124 and determines that it is to generate a preliminary select 1126 for the second byte compressed in the second token. The preliminary select output 1128 for output byte 3 is index = (i1 – 3).

In this decompression cycle, all output bytes have been used to generate preliminary selects. However, some of the data represented by the second token and all of the data represented by the third token are not decompressed in this compression cycle. Decompression of these tokens will be completed in one or more subsequent decompression cycles.

In this example, the preliminary selects 1128 are examined by stage three 1130, and final selects 1132 are output to stage four 1134. If a preliminary select 1128 for an output byte has an overflow bit set, then the preliminary select is resolved by copying the final select from a previous output byte to the output byte to be used as the final select for the output byte. If the overflow bit for a preliminary select 1128 is not set, then the preliminary select 1128 is passed through stage three 1134 as the final select 1132 for the output byte.

In stage one, count and token size information for the tokens loaded in the decompression cycle may be examined in Used Data Calculation logic 1123. If one or more tokens have been completely decompressed, then the total number of bits of the tokens is used to shift the compressed data to align the next input bytes 1120 for the next decompression cycle. A count of the number of output bytes generated from a partially processed token may be used in stage one 1122 in the next decompression cycle to

determine which byte represented in the partially processed token is the first byte not decompressed in the previous decompression cycle. In the example shown in Figure 42b, the first token was completely decompressed in the decompression cycle. The size of the first token is 10 bits, so the compressed data may be shifted 10 bits to align the input bytes 1120 for the next cycle. Two of the three bytes represented by the second token were decompressed in the decompression cycle, so a byte count of 2 is used in the next decompression cycle to continue decompression of the second token.

When the next decompression cycle starts, tokens are extracted from the newly aligned input bytes 1120 and loaded in the decoders for the cycle. In this example, the second token, loaded in decoder 1 in the first decompression cycle, is loaded in decoder 0 in the new decompression cycle. The third token, loaded in decoder 2 in the first decompression cycle, is loaded in decoder 1 in the new decompression cycle. If the next token in input bytes 1120 is a complete token, it will be loaded in decoder 2 for the new decompression cycle. In the new decompression cycle, a preliminary select 1128 will be generated for output byte 0 for the third byte compressed in the second token. A preliminary select 1128 will be generated for output byte 1 for the data byte in the third token. If there is a token being decompressed in decoder 2, then a preliminary select 1128 will be generated for output byte 2 for the first byte compressed in the token. If the token being decompressed in decoder 2 represents more than one compressed bytes, then a preliminary select 1128 will be generated for output byte 3 for the second byte compressed in the token.

If a token being decoded in decoder 0 represents N uncompressed bytes, and the decompression engine can decompress at most M output bytes in a cycle, then the token can be fully decompressed in N/M decompression cycles, wherein N/M is rounded up to the next highest integer if N is not evenly divisible by M. In the embodiment illustrated in Figure 42b, M = 4. A 25-bit token, as illustrated in Figure 32, can represent up to 4096 symbols. In the embodiment illustrated in Figure 42b, it will take 4096/4 = 1024 cycles to fully decompress the token. If a token representing N uncompressed bytes is partially decompressed in a decompression cycle, then in some cases it may take N/M + 1 cycles to decompress. For example, in the embodiment of decompression engine 550 illustrated in

Figure 33, there are 8 input bytes (64 bits), 8 decoders, and 16 output bytes. If the 25-bit token representing 4096 symbols is initially loaded in decoder 0, it will take 4096/16 = 256 cycles to fully decompress the token. If the token is initially loaded in decoder 1, and a token loaded in decoder 0 represents less than 16 symbols (for example, 8), then the first 8 symbols from the token in decoder 1 will be decompressed in a first cycle. The token will be loaded in decoder 0 in the second cycle. The remaining 4088 symbols represented by the token will be decompressed in 4088/16 = 256 cycles (the fraction is rounded up). Thus, it will take 257 cycles to fully decompress the token.

In one embodiment, as a token is being decompressed over multiple cycles, the remaining output symbols to be generated may be output to the other decoders in stage one and to Used Data Calculation 1123. This may prevent the other decoders from decoding tokens until there are output bytes available, and may also prevent the input data from being shifted until the token is completely decompressed. In some embodiments, any number larger than the maximum number of output bytes may be output by a decoder to signal that the token will not complete decompression in this cycle to save output bits. For example, in the embodiment illustrated in Figure 42b, a 5 might be output by decoder 0 to indicate that the token loaded in decoder 0 will not be completely decompressed in the current decompression cycle. Outputting a 5 takes 3 bits, while outputting a 4096 would take 12 bits.

Figures 43a-43k – Flowcharts describing a parallel decompression engine

Figures 43a-43k illustrate flowcharts describing embodiments of parallel decompression processing in embodiments of decompression engine 550.

Figure 43a - The operation of a parallel decompression engine

Figure 43a is a high-level flowchart illustrating an embodiment of decompression processing in an embodiment of parallel decompression engine 550. Parallel decompression engine 550 receives compressed data 900 to be decompressed, and outputs uncompressed data 970. Compressed data 900 is a compressed representation of uncompressed data 970. Compressed data 900 may comprise one or more tokens. Each

token in compressed data 900 may be an encoded description of one or more uncompressed symbols in uncompressed data 970. Compressed data 900 may have been compressed by any of a variety of compression methods, including, but not limited to parallel and serial compression methods. Figures 43b-43k illustrate the flowchart of Figure 41a in greater detail

Figure 43b - A parallel decompression method

Figure 43b illustrates an embodiment of a parallel decompression method performed in one embodiment of the parallel decompression engine 550 of Figure 43a. Figure 43b illustrates that compressed data may be decompressed in a series of cycles, with one or more tokens from the compressed data examined and decompressed in parallel in each cycle. In block 906, the parallel decompression engine may examine a plurality of tokens from the decompressed data. The plurality of tokens may be examined in parallel, i.e., more than one token may be examined at a time. If it is determined in block 906 that all tokens in the compressed data have been decompressed by the decompression engine, then in block 932 the decompression process may stop. If it is determined in block 906 that there are tokens to be examined and decompressed, then the tokens are examined, and information extracted from the tokens in block 906 may be passed to block 934. In one embodiment, the information extracted from the tokens is passed to block 934 in parallel.

In block 934, the information extracted from the tokens in block 906 may be used to generate a plurality of selects, or pointers, that point to symbols in a combined history window. The combined history window may include uncompressed symbols from previous cycles of the decompression engine. The portion of the combined history window comprising uncompressed symbols from previous decompression cycles may be referred to as the history window or history table. The combined history window may also include uncompressed symbols from the current decompression cycle. The uncompressed symbols from the current decompression cycle may be referred to as "data bytes." During compression, one or more uncompressed symbols may not be compressed, and may be stored in a token in uncompressed form. The decompression engine recognizes tokens comprising uncompressed symbols, extracts the uncompressed symbols from the tokens,

and passes the uncompressed symbol to the combined history window unchanged. Thus, selects generated in block 934 may point to either uncompressed symbols from previous decompression cycles or uncompressed symbols from the tokens being decompressed in the current cycle.

In block 954, the decompression engine uses the selects generated in block 934 to extract the one or more uncompressed symbols pointed to by the selects from the history window, and copies the extracted uncompressed symbols to uncompressed output data 970. The uncompressed symbols may be appended to the end of output data 970. Output data may be an output data stream, i.e., the data may be streamed out to a requesting process as it is decompressed, or alternatively the output data 970 may be an uncompressed output file that is not released until the entire compressed data 900 is decompressed.

In block 960, the uncompressed symbols from the current decompression cycle may be written to the history window. If the history window is full, one or more of the oldest symbols from previous decompression cycles may be moved out of the history window prior to writing the uncompressed symbols from this decompression cycle. The oldest symbols may be shifted out of the history window, or alternatively the history window may be a "ring buffer," and the oldest symbols may be overwritten by the new symbols. Figures 43c-43k illustrate the flowchart of Figure 43b in greater detail

Figure 43c - Examining a plurality of tokens in parallel

Figure 43c expands on block 906 of Figure 43b, illustrating one embodiment of a method for examining a plurality of tokens from the compressed data 900 in parallel. In block 908, one or more tokens to be decompressed in parallel in the current decompression cycle may be extracted from the compressed data 900. The tokens may be extracted from the compressed data beginning at the first token compressed by the compression engine that compressed the data, and ending at the last token compressed by the compression engine. A maximum number of tokens may be decompressed in one cycle. As an example, the decompression logic illustrated in Figure 33 accepts a maximum of eight tokens in a decompression cycle. Preferably, a decompression engine may accept less than the maximum number of tokens in a decompression cycle. Thus, the decompression logic

illustrated in Figure 33 accepts a minimum of one token in a decompression cycle, for example, in a last decompression cycle when only one token is left to decompress. If a token represents more uncompressed output symbols than can be compressed in a decompression cycle, then it will take more than one decompression cycle to fully decompress the token. Information in the token may be used in extracting the token. For example, the size of the token and the number of symbols to be decompressed by the token may be used in extracting the token. In one embodiment, the size of a token may be the size in bits of the token. Figure 43d illustrates one embodiment of a process for extracting tokens in greater detail.

In block 924, the tokens extracted for this decompression cycle may be examined in parallel, and information about the tokens may be generated for use in the decompression cycle. Examples of information that may be extracted from a token include, but are not limited to: a count representing the number of uncompressed symbols this token represents; data byte information; and index information. Data byte information may include an uncompressed symbol if this token represents a symbol that was not compressed by the compression engine. Data byte information may also include a data byte valid flag indicating that the data byte for this token is valid. In one embodiment, the data byte valid flag may be a bit that is set (1) if the data byte is valid, and not set (0) if the data byte is not valid. Index information may include an index. In one embodiment, the index may represent an offset from the position in the uncompressed data 970 to receive first uncompressed symbol to be decompressed from the information in this in this token to the first uncompressed symbol previously decompressed and stored in the uncompressed data 970 to be copied into the position. In one embodiment, the previously decompressed symbols from one or more decompression cycles may be in a history window, and the maximum value for the index may be related to the length of the history window. In one embodiment, the index valid flag may be a bit that is set (1) if the index is valid, and not set (0) if the index is not valid. Figure 43e illustrates one embodiment of a process for generating information from tokens in parallel in greater detail.

Figure 43d - Extracting one or more tokens to be decompressed in parallel

Figure 43d expands on block 908 of Figure 43c, and illustrates one embodiment of a method for extracting one or more tokens to be decompressed in parallel from compressed data 900. In block 910 of Figure 43d, the method determines if there is more input data, i.e., if more tokens remain in the compressed data 900 to be decompressed. If so, then in block 912 the method determines if a decoder is available. If a decoder is not available, then all decoders have been assigned tokens to be decompressed, and the decompression cycle continues in block 924 of Figure 43c.

If a decoder is determined to be available in block 912, then the method may proceed to blocks 914 through 920. Blocks 914 through 920 may determine how much of the compressed data 900 to use in the current decode, and also may determine how many decoders to use in the current decode. In one embodiment, blocks 914 through 920 may be performed in stage one of the decompression engine illustrated in Figure 33. In block 914, the method may determine the size of a token representing compressed data. In block 915, the method may examine the token to see if it is a complete token. If the tokens are being loaded in the decoders from a section of the compressed data, for example a 32-bit section, then, after extracting at least one token, the remaining bits in the input data may not comprise an entire token. The size of the token determined in block 914 may be compared to the number of bits left in the input data to determine if there is a complete token. If the token is not complete, then the method may continue to block 924 of Figure 43c.

In block 916, the method may determine the number of symbols that will be generated by the decompression of this token. In block 918, the method may shift the input data by the size of the token to make the next compressed token in the compressed data 900 available to be extracted by this process. The shifting of the input data may not occur until the decompression cycle determines how many tokens will be fully decompressed in this cycle, and the data may be shifted by the total size in bits of all tokens fully decompressed in this cycle. The shifting may prepare the input data for the next decompression cycle. In block 920, the method may determine if more symbols will be decompressed by the tokens to be decompressed in this decompression cycle (counting the current token being examined) than the maximum output width for one decompression cycle. The maximum number of uncompressed symbols that may be decompressed in one cycle minus the

number of uncompressed symbols to be produced by the decompression of tokens already extracted for this decompression cycle yields the maximum number of symbols that may be decompressed from the token currently being examined. If the output width has been met or exceeded, then the decompression cycle may continue without the current token being examined being assigned to a decoder. In one embodiment, a token may be partially compressed in a decompression cycle to insure that a maximum number of symbols are decompressed in the cycle. The first token not fully decompressed will be the first token extracted in the next decompression cycle. If the output width has not been met or exceeded as determined in block 920, then the method returns to block 910, and blocks 910-920 may be repeated until there is no more data, or until the output width is met or exceeded.

In block 922, if there is no more input data as determined in block 910, but one or more tokens have been assigned to decoders for decoding, then the decompression cycle continues with block 924 of Figure 43c. This covers the case when there are no more tokens in the compressed data 900, but one or more tokens have been assigned to decoders in blocks 910-920. In block 922, if there is no more input data as determined in block 910, and no tokens have been assigned to decoders, the decompression of the compressed data is complete, and decompression stops.

Figure 43e - Generating count and index or data byte information in parallel

Figure 43e expands on block 924 of Figure 43c, and illustrates one embodiment of a process for generating information from a plurality of tokens in parallel. Illustrated are several items that may be extracted from one or more tokens being decoded in parallel in the current decompression cycle by decoder logic similar to that illustrated in Figure 34.

In block 926 of Figure 43e, a count may be generated for each token being decoded in the current decompression cycle. The count for a token may represent the number of uncompressed symbols the decompression of the token will produce. The count for a token may be between one and the maximum number of symbols that can be represented by a token. For example, in the table of Figure 32, a 25-bit token can represent up to 4096

uncompressed symbols. The count for a token representing an uncompressed symbol will be 1.

In block 928, index information may be generated for each token being decoded in the current decompression cycle. The index information may include an index for one or more tokens being decompressed and an index valid flag for each token being decompressed. A valid index may be generated for a token if the token represents one or more compressed symbols. In one embodiment, the index may represent a distance in symbols from the destination position in the uncompressed data 970 for the first uncompressed symbol to be decompressed from this token to a first uncompressed symbol previously decompressed and stored in the uncompressed data 970. In one embodiment, the previously decompressed symbols from one or more decompression cycles may be stored in a history window, and the index may be an offset to a previously uncompressed symbol in the history window. In one embodiment, the index valid flag may be a bit that is set (1) if the index is valid, and not set (0) if the index is not valid. The index valid flag may be set for tokens for which an index is generated. In one embodiment, the index valid flag may be a bit that is set (1) if the index is valid, and not set (0) if the index is not valid.

In block 930, data byte information may be generated for one or more tokens being decoded in the current decompression cycle. Data byte information for a token may include an uncompressed symbol (data byte) if this token represents a symbol that was not compressed by the compression engine. Data byte information may also include a data byte valid flag indicating that the data byte for this token is valid. In one embodiment, the data byte valid flag may be a bit that is set (1) if the data byte is valid, and not set (0) if the data byte is not valid.

Figure 43f - Generating a plurality of selects to symbols in a combined history window

Figure 43f expands on block 934 of Figure 43b, and illustrates one embodiment of a process for generating in parallel a plurality of selects to symbols in a combined history window. In block 936, one or more preliminary selects may be generated using the information generated in block 924 for this decompression cycle. A preliminary select may be generated for each of the symbols being decompressed in the current decompression

cycle. In one embodiment, a preliminary select is an adjusted index with a single bit overflow. The index is adjusted by an offset from a starting index of a string of symbols in previous uncompressed symbols. The size of the preliminary select is determined by the combined size of the history window, the maximum number of data bytes (determined by the number of decoders), and the overflow bit. For example, for a 64-entry history window, plus eight data bytes, plus a single overflow bit, a preliminary select may be a minimum of eight bits. In this example, the selects may have the values of 0-63 if a window value is to be used for this output symbol or the values of 64-71 if one of the eight data bytes is to be used for this output symbol. The overflow output bit may be set if the data for the output symbol is being generated by one or more of the other tokens being decoded in this decompression cycle. Other combinations of bits may be used to signal to the later stages that no data is being generated for this output symbol in this decompression cycle.

In one example of a decode where an overflow bit may be set, a first decoder may decode a first token and output a pointer to a first data byte, and a second decoder may decode a second token and output a pointer to a second data byte. A third decoder may decode a third token that represents a compressed string including the first and second data bytes generated from the first and second tokens. As these data bytes are not in the history window yet, the overflow bit 26008 is set to signify that the data for the third decoder's output byte is defined by one of the prior decoders in the current decode. The preliminary select output of the second stage for the third decoder is resolved into a final select in the third stage. In this example, two final selects may be generated for the third token; the first pointing to the first decoder's data byte, and the second pointing to the second decoder's data byte.

Figure 43g - Generating preliminary selects

Figure 43g expands on block 936 of Figure 43f, and illustrates one embodiment of a process for generating preliminary selects to symbols in a combined history window. A preliminary select may be generated for each of the output symbols using the information generated in block 924 in the current decompression cycle. In block 938, preliminary selects to symbols in the history window may be generated. For example, if the history

window includes 64 entries indexed 0-63, with 0 being the most recent entry, then, for an output symbol to be copied from the eighth most recent entry in the history window, an index of 7 would be generated.

In block 940, preliminary selects to data bytes in the combined history window may be generated. For example, the history window includes 64 entries indexed 0-63, and the combined history window includes eight data bytes passed from eight decoders in stage one, the eight data bytes may be indexed as data bytes 64-71. For an output symbol to be copied from the third data byte, an index of 66 would be generated.

In block 942, preliminary selects to symbols being generated in the current decompression cycle may be generated. In other words, the symbols required to uncompress the output symbol are not in the history window yet, but are being generated by prior output symbols in this decompression cycle. For these preliminary selects, an overflow bit is set to indicate that the preliminary select needs to be resolved. The index generated for the preliminary select indicates which of the prior output symbols in this decompression cycle contains the symbol required by this output symbol. For example, if there are four output symbols 0-3, and this is the third output symbol (output symbol 2), then, if the overflow bit is set, the index may indicate that the data for this output symbol is being generated on output symbol 0 or 1, but not on output symbol 3.

Figure 43h - Generating final selects

Figure 43h expands on block 944 of Figure 43f, and illustrates one embodiment of a process for generating final selects to symbols in a combined history window. A final select may be generated for each of the output symbols using the information generated in block 924 in the current decompression cycle. In block 946, the overflow bit of each of the preliminary selects may be examined. If the overflow bit is not set, the preliminary select may be passed through unmodified as the final select for the output symbol. If the overflow bit is set, then the preliminary select is resolved. In one embodiment, the preliminary select for this symbol and the final select from each prior output symbol is passed as input to the preliminary select resolution logic for each output symbol. If the preliminary select for an output symbol needs to be resolved, then the index passed in the preliminary select for the

output symbol is used to generate the number of the prior output symbol which will contain the data for this output symbol. The final select for the prior output symbol is then passed through as the final select for this output symbol. For example, if there are four output symbols 0-3, and the overflow bit is set for the third output symbol (output symbol 2), then, if the index indicates that the data for this output symbol is being generated on output symbol 1, the final select from output symbol 1 is copied and passed through as the final select for output symbol 2. The final select from output symbol 1 may be an index to either a symbol in the history window or to a data byte.

Figure 43i - Writing uncompressed symbols to the output data

Figure 43i expands on block 954 of Figure 43b, and illustrates one embodiment of a process for writing the symbols for the output bytes to the uncompressed output data. In block 956, the final selects indexing data bytes passed from the decoders may be used to locate the data bytes and copy the uncompressed data bytes into the output data. In block 958, the final selects indexing symbols in the history window may be used to locate the uncompressed symbols and copy the symbols into the output data. The output symbols may be assembled in the output data in the order of the output symbols in the decompression engine. For example, if there are 16 output symbols (0-15) being generated in a decompression cycle, output symbol 0 may be the first in the output data, and output symbol 15 may be the last. A decompression cycle may not generate a full set of output symbols. For example, with the16 maximum output symbols in the previous example, a decompression cycle may generate only nine output symbols (output symbols 0-8). Preferably, every decompression cycle decompresses as close to the maximum number of output symbols as possible. Some decompression cycles, for example, the last decompression cycle, may not generate the maximum number of output symbols.

Figure 43j - Writing symbols to the history window

Figure 43j expands on block 960 of Figure 43b, and illustrates one embodiment of a process for writing the symbols uncompressed in a decompression cycle to the history window. In one embodiment, the history window may be set up as a buffer, and the oldest

data may be shifted out to make room for the newest data. In another embodiment, the history window may be set up as a ring buffer, and the oldest data may be overwritten by the newest data. Blocks 962 and 964 assume the oldest data may be shifted out of the history window, and may not be necessary in embodiments using a ring buffer for the history window.

In block 962, the history window is examined, and if there is not enough room for the symbols decompressed in this cycle, in block 964 the data in the history window is shifted to make room for the new data. In one embodiment, the history window may be shifted after every decompression cycle to make room for the new data.

In block 966, the newly uncompressed symbols are written to the end of the history window. In one embodiment, the symbols may be written to the history window using the method described for writing the symbols to the output data described for blocks 956 and 958 of Figure 43i.

Figure 43k - A decompression process combining Figures 43b, 43c and 43d

In Figure 43k, several of the blocks from Figures 43a-43j are combined to further illustrate one embodiment of a decompression cycle. Blocks 910-922 are from Figure 43d and expand on block 908 of Figure 43c, illustrating one embodiment of a method for extracting one or more tokens to be decompressed in parallel from the input compressed data as described for Figure 43d. In block 924, the tokens extracted for this decompression cycle may be examined in parallel, and information about the tokens may be generated for use in the decompression cycle. The operation of block 924 is described in Figures 43c and 43e. In block 934, the information extracted from the tokens may be used to generate a plurality of selects, or pointers, that point to symbols in a combined history window. The operation of block 934 is described in Figures 43b, 43f, 43g, and 43h. In block 954, the decompression engine uses the selects generated in block 934 to extract the one or more uncompressed symbols pointed to by the selects from the history window, and copies the extracted uncompressed symbols to uncompressed output data. The operation of block 954 is described in Figures 43b and 43i. In block 960, the uncompressed symbols from the

current decompression cycle may be written to the history window. The operation of block 954 is described in Figures 43b and 43j.

After writing the uncompressed symbols to the history window, operation may return to block 910 to determine if there is more input data available. If there is no more input data available as determined in block 910 and there are no valid decodes as determined in block 922, then operation completes. Otherwise, the next parallel decompression cycle begins.

Decompression Timing

Each stage in this design has been timed to achieve 100MHz with 0.25μ technology and low power standard cell design library. Alternate embodiments may use custom data-paths or custom cells to achieve higher clock rates or fewer stages. Stage 1 25501 has proven to be the most critical at 9.1nS in standard cell design. Stage 2 25505, required only 3.8nS, with stages 3 25509 and 4 25513 at 8.23nS and 1.5nS respectively. There will be some additional powering logic delay in stage 4 not accounted for in these calculations, which are not a problem due to the timing margin of stage 4 25513.

Scalable Compression / Decompression

The IMC 140 also includes scalable compression / decompression, wherein one or more of the parallel compression / decompression slices can be selectively applied for different data streams, depending on the desired priorities of the data streams.

Concurrency

The IMC 140 also allows concurrency of operations by allocation of multiple data requests from a plurality of requesting agents or from multiple data requests input from a single requesting agent. On average, when the compression and decompression unit 251 is used, the requested data block is retired sooner than without use of the current invention. When multiple data requests are queued from concurrent sources, the pending transactions can complete with less latency than in prior art systems. As the input block size grows and

the number of pending concurrent data requests increase, the present invention becomes increasingly attractive for reduction of latency and increased effective bandwidth.

Figures 44a and 44b - Memory Module Embodiment

Figures 44a and 44b show a board assembly drawing of one embodiment of a memory module 571 which includes the MemoryF/X Technology. As shown, the memory module 571 includes a plurality of memory devices 573 as well as a MemoryF/X Technology Compactor chip 250. The MemoryF/X Technology Compactor chip 250 may include only a subset or all of the MemoryF/X Technology. For example, the MemoryF/X Technology Compactor chip 250 may include only the parallel compression / decompression engine portion of the MemoryF/X Technology for in-line real time compression. The MemoryF/X Technology Compactor chip 250 may also include virtual memory logic for implementing improved virtual memory functions using the parallel compression / decompression technology described herein.

Figure 44a illustrates the front side of the module and Figure 44b illustrates the back side of the module. Figures 44a and 44b illustrate a currently preferred embodiment of the memory module design, which is preferably a 256MB registered DIMM, which is compliant with the Intel PC100 or PC133 specification. Alternatively, other embodiments may be designed for larger and/or smaller registered DIMMs or different form factors or specifications. The MemoryF/X Technology 200 may of course be included in other memory module designs. Additionally, the MemoryF/X Technology 200 or variations of the MemoryF/X Technology 200 may be used with Rambus or Double Data Rate DRAM devices. Other alternate embodiments may include different DRAM population options, memory types such as those proposed in the JDEC standard. In addition, alternate embodiments may include a mix of these memory types on multiple different memory module standards.

Memory Module Compression

The following describes methods to improve the performance of memory intensive, I/O bound applications by utilizing high-bandwidth compression hardware integrated on industry standard memory modules that can be plugged into general computing systems.

The present invention includes compression technology, referred to either as the Memory F/X technology or as the GigaByte Compression™ technology, which is capable of compressing data at rates comparable to main memory bandwidths, it can be added to the main memory subsystem and accessed without decreasing the overall available bandwidth to the memory subsystem. Ideally, the compression hardware would be located inside the memory controller or processor. Until then, adding it to a memory module, such as a SDRAM DIMM, will prove to be very effective. The processor uses the compression hardware located on a SDRAM DIMM to compress and decompress database pages between a compressed buffer cache and uncompressed buffer cache. The processor should be able to transfer pages in about 15us or up to 75K pages per second that is more than an order of magnitude faster than compression hardware located on the PCI bus.

The preferred embodiment is an SDRAM DIMM that includes normal SDRAM and our compression hardware. The compression hardware comprises a compression engine, decompression engine and enough buffer space to ensure that back-to-back 4KB pages can be either compressed or decompressed at full SDRAM DIMM bandwidth. Preferably, the compression and decompression engines are contained in a single chip that interfaces directly to the SDRAM DIMM interface along with the SDRAMs. If necessary, separate buffers and transceivers may be added for electrical timing reasons. The memory module behaves just like a normal SDRAM DIMM. The compression hardware is added to the SDRAM DIMM without changing any protocols or affecting the electrical timings. BIOS accesses the memory module SPD bus, initializes the memory controller, and tests the SDRAM just like it always does. During the boot process, at least one RAS page of main memory, mapped to the SDRAM DIMM, is allocated for subsequent use by our drivers to access the compression and decompression engines. During the boot process, our drivers are loaded and the compression hardware is initialized. Our device drivers perform a

unique sequence of accesses to the allocated pages in main memory belonging to the compression hardware. The unique sequence contains a key and a command. The compression hardware snoops all accesses to the SDRAMs. When the compression hardware decodes the key, it decodes the command that follows. The command indicates whether the compression hardware is to be enabled or disabled permanently. If the driver instructs the compression hardware to be disabled, then the pages allocated to the compression hardware can be deallocated and used by the system just like normal. If the compression hardware is enabled, then operating system drivers and application software can begin accessing the compression hardware through our device drivers. Read and write accesses to the compression hardware also flow through to the SDRAMs. In this way, the compression hardware is non-intrusive and does not affect the SDRAM DIMM interface protocols. During read operations, both the SDRAM and compression hardware try to return data to the SDRAM interface. During RAS, the compression hardware decodes the RAS address to determine who is being accessed. If the compression hardware is selected, then the compression hardware disables the output enables of the SDRAMs.

The compressed buffer cache can be located anywhere in coherent main memory. The size of the compressed cache is only limited by the amount of main memory in the system and the software that manages it. Compressed pages are transferred between the I/O subsystem and compressed buffer cache by a DMA controller. Pages are copied from one location in main memory to another by the processor by reading the page from compressed memory, writing the page into the compression engine, reading the uncompressed page from the compression engine, and writing the page to uncompressed main memory.

If the memory DIMMs are interleaved on at least a page boundary, and a compression engine exists on each DIMM, then a more enhanced transfer protocol can be used. Before decompression a page, the address of the page to be decompressed is written into the compression engine. When the processor reads the compressed page from the compressed buffer cache, the compression engine automatically snoops the data and decompresses the page on its own. The processor reads the uncompressed data from the decompression engine and writes it to uncompressed main memory. As a result, the processor should be able to transfer pages in about 12.5us or up to 80K pages per second.

One of the negative side effects associated with lossless compression is that occasionally a page is compressed and the result is larger than the original uncompressed page. This embodiment allows us to write a page into the compression hardware, determine whether the page compressed, and optionally read back the uncompressed page depending on whether or not it compressed.

Modifying the Memory Controller

If the memory controller is modified to allow more flexible access protocols to memory modules, such as adding wait states, or supporting a non-deterministic protocol, then it would be possible to put the compression engine in-line with main memory. For this case, the compression hardware is still used to compress and decompress database pages to and from the compressed buffer cache of the database server. However, the method used to transfer pages from compressed memory to uncompressed memory is transparent to the memory copy routines. In this embodiment, the compression engine and management hardware are located in-line with main memory. Before decompressing a page, the processor writes the address of the page to be decompressed into the compression engine. When the compressed page is read from compressed main memory, it is automatically decompressed real-time. The processor follows by writing the page to uncompressed main memory. The processor should be able to compress and decompress pages at the same rate as a normal memory copy.

Memory Module Compression Applications

There are various ways that an operating system or application software can take advantage of memory module compression. A few of the ways will be explored. For this discussion, it is assumed that the system includes compressed disks, and main memory is partitioned into compressed buffer cache, uncompressed buffer cache, and uncompressed application space.

Using a DMA controller, compressed pages are transferred from local and remote disks to compressed buffer cache, or they can be decompressed and transferred directly to uncompressed buffer cache. In general, memory module compression is used to compress

and decompress 4KB pages between compressed buffer cache and uncompressed buffer cache and application space. To realize the full benefits of memory module compression, it is assumed that both code and data may be compressed on disk. However, code only represents a small fraction of overall main memory usage.

One of the key parameters used to tune a database server is by varying the size of the database buffer cache. It is common to dedicate more than half of system memory to the buffer cache.

Operating Systems

An operating system can use memory module compression in a general-purpose way that is transparent to application software. From an operating system perspective, the compressed buffer cache could reside between the file system and the disk and network device drivers. For example, the compressed buffer cache could be inserted into Windows 2000 using a file system filter driver. Alternatively, the compressed buffer cache could reside between the file system and the uncompressed buffer cache. For example, the filter system and buffer cache managers associated with Windows 2000 could be replaced with a compressed file system and compressed buffer cache managers.

Application Software

Application software can use memory module compression in an application specific way. User-mode compressed file systems and compressed buffer cache managers can be incorporated directly into applications eliminating the need to make expensive kernel-mode calls. It is common for database software to incorporate their own file systems and buffer cache managers based on using raw I/O. From a software perspective, the compressed cache could reside between the database file system and the disk and network device drivers. Alternatively, the compressed buffer cache could reside between the file system and the uncompressed buffer cache.

In all cases, as long as the compressed page transfer rates are slower than uncompressed page transfer rates, the uncompressed buffer cache will continue to be viewed as being higher in the memory hierarchy than the compressed buffer cache. As

these transfer rates become equal, the distinction between compressed and uncompressed will disappear. There will only be compressed buffer cache. However, for those applications that allow processes to access data directly from buffer cache rather than copying the page to uncompressed application space, there will continue to be a need for an uncompressed buffer cache since random accesses into compressed pages statistically have very long access times.

Page Faults

For example, when a user process accesses a page that does not exist in the applications user space, a page fault occurs and control is switched to a supervisor-mode page fault handler. The virtual memory manager attempts to locate the page in uncompressed buffer cache. If it is not present, then the virtual memory manager allocates a page in buffer cache for the new page. Most likely, an old page must be deallocated from buffer cache to make room for the new page. The virtual memory manager accesses the file system to determine the location of the page within the I/O subsystem and generates a disk request. When the page returns, it is transferred to the buffer cache. Finally, the buffer cache page is mapped into user space. Control is finally returned to the user process and process execution resumes.

File System Filter

If the compressed buffer cache resides between the file system and uncompressed buffer cache, then the compressed cache is searched as soon as it can be determined that the page is not in uncompressed cache. Furthermore, if the compressed and uncompressed buffer caches are managed together, then the location of the page can be determined even faster. Once it has been determined that the page does not exist in either buffer cache, the file system is accessed. Since the compressed buffer cache manage accesses compressed page from disk, the file system is replaced by a compressed file system that can handle varying memory capacity. The compressed file system maintains compressed page allocation tables. The compressed page allocation tables are searched to determine where

the compressed pages reside within the I/O subsystem. The compressed file system generates the appropriate raw I/O requests.

Compressed Buffer Cache

If the compressed buffer cache resides between the file system and disk and network device drivers, then the compressed cache can not be searched until it can be determined that the page is not in uncompressed cache and the file system is accessed. Once the file system is accessed, logical raw I/O requests are generated. The logical raw I/O requests are intercepted by the compressed buffer cache manager. The compressed buffer cache is searched. If the page is already the compressed buffer cache, then the page is decompressed and transferred directly to uncompressed buffer cache. If the page does not reside in the compressed buffer cache, then the compressed page allocation tables are searched to determine where the compressed pages reside within the I/O subsystem. The compressed buffer cache manager generates the appropriate raw I/O requests.

Buffer Cache Management

The most frequently accessed pages should be maintained in uncompressed cache. If all data fits in memory, then no data is compressed. When data size exceeds available memory size, then the least frequently used (not necessary the least recently used) pages are compressed until enough space is available for the new page. Unnecessary movement of pages between the uncompressed and compressed caches can actually cause performance to drop. A buffer cache manager that manages both the compressed and uncompressed buffer caches is more effective because the relative cache sizes and caching algorithms used to managed these caches can be designed to adapt to varying workload behavior. It is more difficult, if not impossible, to adapt a separate compressed buffer cache, such as a file system filter, and uncompressed buffer cache to varying workload behavior.

There are various ways to efficiently manage the compressed and uncompressed buffer caches. To minimize main memory use, and minimize main memory accesses, the contents of the buffer caches should be mutually exclusive. If a page exists in compressed buffer cache, then it should not exist in uncompressed cache, and visa versa. From a

hierarchy perspective, the uncompressed buffer cache is the primary cache, and the compressed buffer cache is the secondary cache. The replacement algorithm for both caches should be based on which pages are least recently used, as well as least frequently used.

Compacting Pages

Pages that are frequently accessed together can be detected, compressed, and compacted into as few compressed blocks as possible.

Memory Module Compression Management

When uncompressed memory pages of known size are compressed, they become variable sized compressed pages. These compressed blocks must be managed with very little overhead in terms of access time and additional memory space in order to realize any benefits from compression. Typical software-based compression management efficiencies are 90% or better, where allocation and deallocation takes only a few memory accesses. Compression management schemes can range from all software-based to all hardware-based implementations. Hardware-based compression management has the advantage of eliminating the extra memory accesses required to read the compressed page from main memory and requires minimal memory space overhead since the compressed block pointers are built into the compressed pages. In addition, hardware compression management operates with minimal software intervention. Software only has to intervene at a higher level due to issues associated with varying memory capacity.

The type of compressed memory management to use is also influenced by where the compressed pages are stored. If the pages are stored in main memory, such as a compressed buffer cache, then either software management or hardware management, or a combination of the two, is reasonable. If the pages are stored on disk, then a software management scheme is preferred based on compressed page allocation tables stored on disk but cached in main memory.

The secondary compressed buffer cache manager does not access the compressed cache directly. Instead, the compressed cache appears to be uncompressed to the compressed cache manger.

The compression management hardware provides a means to extract a compressed page from the compressed buffer cache without decompressing the page. This is usually performed when preparing to write the compressed page back to disk. The compression management hardware may actually store the compressed page in several smaller compressed blocks scattered across main memory.

Typical Database System

Database Program

Client/server applications typically embed SQL statements or ODBC calls to communicate with database systems. These applications can be a C program, CGI script, etc. Client/server interaction is based on remote procedural calls (RPCs). These client/server applications maintain their own data spaces, separate from the database.

Database Software

The database middleware provides applications with an interface to database systems. The most common database APIs on PC platforms are SQL and ODBC. The database middleware translates SQL statements and ODBC calls embedded in an application into the appropriate database queries and operations. A SQL interpreter is responsible for processing embedded text-based SQL statements received from the application. SQL and ODBC run-time libraries are responsible for processing pre-compiled SQL statements and ODBC calls made by the application. The database engine executes these queries and operations. The database engine includes the query execution engine, optimizer, iterator and sorter. The database engine also maintains a large temporary data space used to carry out these queries and operations. This temporary data space is managed on a page basis. The database engine sends page requests to the buffer cache manager for all data table information needed to carry out the queries and operations. If the buffer cache

manager determines that a requested page is in buffer cache, then the page is copied to the database engine data space. If the page is not in buffer cache, then the page request is sent to the file manager. For performance reasons, commercial databases use their own file system. The file systems provided by the operating system are typically bypassed. The file manager determines were the pages are stored on disk, and routes the appropriate page requests to the device drivers using raw I/O routines.

Operating System

For performance reasons, most commercial databases use raw I/O to access disks. The database may or may not use the raw I/O routines provided by the operating system.

I/O Subsystem

In an uncompressed database system, one or more disks store the database data where each disk contains one or more database-specific partitions. These partitions store the various database files that are accessed on a page basis. A page will contain one or more records from the database, and is organized for easier data management by the software. Normally, one or more of these pages will be stored in a logical sector of the file system. In the ISI Compressed File System, these logical sectors will be processed by the ISI MemoryF/X hardware into Compressed Data Blocks (CDB). A CDB is a variable length entity, and must be managed properly to allow for maximum advantage from its reduced size.

The file system will be enhanced by the ISI compressed block allocation tables to allow these CDBs to be managed in such a way that is transparent to the application. This block management is done such that the translation overhead required is minimized, and the fragmentation of data resulting from small changes in the compressed data size is also minimized. In addition, the advantages of compression will be enhanced by managing these blocks to reduce the number of disk seeks required, and to reduce the total amount of data transferred from the disk to satisfy a request. Free space in the file system will also be managed by the ISI allocation tables to ensure the best use of the physical disk partitions available to the database.

As a result, the compressed data block will be readily managed such that the applications have access to individual logical sectors as before, but the resulting sectors are stored as compressed blocks. The ISI allocation tables will also ensure that all compressed block allocation reside on the physical media that contains the compressed data in a manner that ensures reliability and transportability of the database.

Average Seek Time Reduction

An important metric in the performance of a database system is the Average Seek Time for data being accessed from the I/O subsystem. This metric is based on the effectiveness of any disk cache that may be in use, and the actual seek time of the hard drives being accessed. If an overall compression ratio of 2:1 is achieved for a particular database, only half the amount of data will be requested from the disk for a query. This result could cut the number of disk seeks required to access the data in half.

In an uncompressed file system, the FAT allows sequential sectors to be placed on the drive in any location that is free, possibly requiring multiple seeks for a group of pages. In a compressed system, these sectors are more likely to be placed in a single logical sector or a group of sequential sectors due to their reduced size. This will result in fewer disk accesses for a group of pages. This is especially important as the database matures and the file system becomes more fragmented. But even in a well organized database, the number of FAT to Sector seeks and Track to Track seeks will be reduced, for both fetches and stores, due to the reduction of data volume.

In addition, due to the compression of the data, the effective size of the compressed buffer cache is increased. For an average compression ratio of 2:1, the hit ratio will double, reducing the number of disk seeks sent to the I/O subsystem. To further enhance compressed data management and improve the overall compressed buffer cache hit rate, newly written pages can be grouped together with the expectation that they will be accessed as a group.

Thus, use of the Memory F/X technology increases database performance by the insertion of compressed pages into a compressed buffer cache, effectively increasing the size of buffer cache memory and reducing the number of disk seeks.

Figures 45 - 48 illustrate how Memory F/X can be utilized by various applications. Initially, this technology will be targeted towards high volume Intel-based servers, and eventually migrate toward high margin non-Intel servers that use custom ASIC technology for control of the memory subsystem.

Although the system and method of the present invention has been described in connection with the preferred embodiment, it is not intended to be limited to the specific form set forth herein, but on the contrary, it is intended to cover such alternatives, modifications, and equivalents, as can be reasonably included within the spirit and scope of the invention as defined by the appended claims.